

Mining Tree-Query Associations in Graphs

Eveline Hoekx and Jan Van den Bussche
Hasselt University and transnational University of Limburg
Agoralaan D, 3590 Diepenbeek, Belgium

Abstract

New applications of data mining, such as in biology, bioinformatics, or sociology, are faced with large datasets structured as graphs. We introduce a novel class of tree-shaped patterns called tree queries, and present algorithms for mining tree queries and tree-query associations in a large data graph. Novel about our class of patterns is that they can contain constants, and can contain existential nodes which are not counted when determining the number of occurrences of the pattern in the data graph. Our algorithms have a number of provable optimality properties, which are based on the theory of conjunctive database queries. We propose a practical, database-oriented implementation in SQL, and show that the approach works in practice through experiments on data about food webs, protein interactions, and citation analysis.

1 Introduction

The problem of mining patterns in graph-structured data has received considerable attention in recent years, as it has many interesting applications in such diverse areas as biology, the life sciences, the World Wide Web, or social sciences. In the present work we introduce a novel class of patterns, called tree queries, and we present algorithms for mining these tree queries and tree-query associations in a large data graph. This article is based on two earlier conference papers [17, 20].

Tree queries are powerful tree-shaped patterns, inspired by conjunctive database queries [18]. In comparison to the kinds of patterns used in most other graph mining approaches, tree queries have some extra features:

- Patterns may have “existential” nodes: any occurrence of the pattern must have a copy of such a node, but existential nodes are not counted when determining the number of occurrences.
- Moreover, patterns may have “parameterized” nodes, labeled by constants, which must map to fixed designated nodes of the data graph.
- An “occurrence” of the pattern in a data graph G is defined as any homomorphism from the pattern in G . When counting the number of occurrences, two occurrences that differ only on existential nodes are identified.

Past work in graph mining has dealt with node labels, but only with non-unique ones: such labels are easily simulated by constants, but the converse

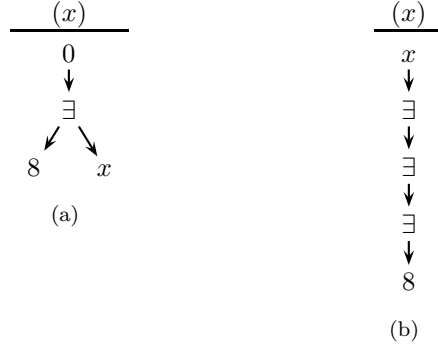


Figure 1: Simple examples of tree-query patterns

is not obvious. It is also possible to simulate edge labels using constants. To simulate a node label a , add a special node a , and express that node x has label a by drawing an edge from x to a . For an edge $x \rightarrow y$ labeled b , introduce an intermediate node $x.y$ with $x \rightarrow x.y \rightarrow y$, and label node $x.y$ by b .

A simple example of a tree query is shown in Figure 1(a); when applied to a food web: a data graph of organisms, where there is an edge $x \rightarrow y$ if y feeds on x , it describes all organisms x that compete with organism #8 for some organism as food, that itself feeds on organism #0. This pattern has one existential node, two parameters, and one distinguished node x . Figure 1(b) shows another example of a tree query; when applied to a food web, it describes all organisms x that have a path of length four beneath them that ends in organism #8.

Effectively, tree queries are what is known in database research as *conjunctive queries* [9, 40, 2]; these are the queries we could pose to the data graph (stored as a two-column table) in the core fragment of SQL where we do not use aggregates or subqueries, and use only conjunctions of equality comparisons as where-conditions. For example, the pattern of Figure 1(a) amounts to the following SQL query on a table $G(\text{from}, \text{to})$:

```
select distinct G3.to as x
from G G1, G G2, G G3
where G1.from=0 and G1.to=G2.from
and G2.to=8 and G3.from=G2.from
```

In the present work we also introduce association rules over tree queries. By mining for tree-query associations we can discover quite subtle properties of the data graph. Figure 2(a) shows a very simple example of an association that our algorithm might find in a social network: a data graph of persons where there is an edge $x \rightarrow y$ if x considers y to be a close friend. The tree query on the left matches all pairs (x_1, x_2) of “co-friends”: persons that are friends of a common person (represented by an existential variable). The query on the right matches all co-friends x_1 of person #5 (represented by a parameterized node), and pairs all those co-friends to person #5. Now were the association from the left to the right to be discovered with a confidence of c , with $0 \leq c \leq 1$, then this would mean that the pairs retrieved by the right query actually constitute a fraction

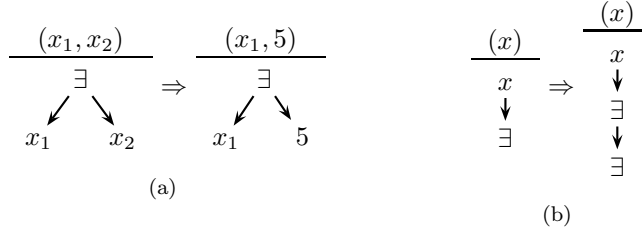


Figure 2: Simple examples of association rules over tree queries.

of c of all pairs retrieved by the left query, which indicates (for nonnegligible c) that 5 plays a special role in the network.¹

Figure 2(b) shows quite a different, but again simple, example of a tree-query association that our algorithm might discover in a food web. With confidence c , this association means that of all organisms that are not on top of the food chain (i.e., they are fed upon by some other organism), a fraction of c is actually at least two down in the food chain.

The examples of tree queries and associations we just saw are didactical examples, but in Section 7 we will see more complicated examples of tree queries and associations mined in real-life datasets.

In this paper we present algorithms for mining tree queries and associations rules over tree queries in a large data graph. Some important features of these algorithms are the following:

1. Our algorithms belong to the group of graph mining algorithms where the input is a single large graph, and the task is to discover patterns that occur sufficiently often in the single data graph. We will refer to this group of algorithms as the single graph category. There is also a second category of graph mining algorithms, called the transactional category, which is explained in Section 2.
2. We restrict to patterns that are trees, such as the example in Figure 1. Tree patterns have formed an important special case in the transactional category (Section 2), but have not yet received special attention in the single-graph literature. Note that the data graph that is being mined is not restricted in any way.
3. The tree-query-mining algorithm is incremental in the number of nodes of the pattern. So, our algorithm systematically considers ever larger trees, and can be stopped any time it has run long enough or has produced enough results. Our algorithm does not need any space beyond what is needed to store the mining results. Thanks to the restriction to tree shapes the duplicate-free generation of trees can be done efficiently.

¹Note that this does not just mean that 5 has many co-friends; if we only wanted to express that, just a frequent pattern in the form of the right query would suffice. For instance, imagine a data graph consisting of n disjoint 2-cliques (pairs of persons who have each other as a friend), where additionally all these persons also consider 5 to be an extra friend (but not vice versa). In such a data graph, 5 is a co-friend of everybody, and the association has a rather high confidence of more than $2/7$. If, however, we would now add to the data graph a separate n -clique, then still $2/3$ of all persons are a co-friend of 5, which is still a lot, but the confidence drops to below $2/n$.

4. For each tree, all conjunctive queries based on that tree are generated in the tree-query-mining algorithm. Here, we work in a levelwise fashion in the sense of Mannila and Toivonen [31].
5. As in classical association rules over itemsets [3], our association rule generation phase comes after the generation of frequent patterns and does not require access to the original dataset.
6. We apply the theory of conjunctive database queries [9, 40, 2] to formally define and to correctly generate association rules over tree queries. The conjunctive-query approach to pattern matching allows for an efficiently checkable notion of frequency, whereas in the subgraph-based approach, determining whether a pattern is frequent is NP-complete (in that approach the frequency of a pattern is the maximal number of disjoint subgraphs isomorphic to the pattern [29]).
7. There is a notion of equivalence among tree queries and association rules over tree queries. We carefully and efficiently avoid the generation of equivalent tree queries and associations, by using and adapting what is known from the theory of conjunctive database queries. Due to the restriction to tree shapes, equivalence and redundancy (which are normally NP-complete) are efficiently checkable.
8. Last but not least, our algorithms naturally suggest a database-oriented implementation in SQL. This is useful for several reasons. First, the number of discovered patterns can be quite large, and it is important to keep them available in a persistent and structured manner, so that they can be browsed easily, and so that association rules can be derived efficiently. Moreover, we will show how the use of SQL allows us to generate and check large numbers of similar patterns in parallel, taking advantage of the query processing optimizations provided by modern relational database systems. Third, a database-oriented implementation does not require us to move the dataset out of the database before it can be mined. In classical itemset mining, database-oriented implementations have received serious attention [39, 36], but less so in graph mining, a recent exception being an implementation in SQL of the seminal SUBDUE algorithm [8].

The purpose of this paper is to introduce tree queries and tree-query associations and to present algorithms for mining tree queries and tree-query associations. Concrete applications to discover new knowledge about scientific datasets are the topic of current research. Yet, the algorithms are fully implemented and we can already show that our approach works in practice, by showing some concrete results mined from a food web, a protein interactions graph, and a citation graph. We will also give performance results on random data graphs (as a worst-case scenario).

2 Related Work

Approaches to graph mining, especially mining for frequent patterns or association rules, can be divided in two major categories which are not to be confused.

1. In transactional graph mining, e.g., [12, 21, 22, 23, 28, 41, 42], the dataset consists of many small data graphs which we call transactions, and the task is to discover patterns that occur at least once in a sufficient number of transactions. (Approaches from machine learning or inductive logic programming usually call the small data graphs “examples” instead of transactions.)
2. In single-graph mining the dataset is a single large data graph, and the task is to discover patterns that occur sufficiently often in the dataset.

Note that single-graph mining is more difficult than transactional mining, in the sense that transactional graph mining can be simulated by single-graph mining, but the converse is not obvious.

Since our approach falls squarely within the single-graph category, we will focus on that category in this section. Most work in this category has been done on frequent pattern mining, and less attention has been spend on association rules. We briefly review the work in this category next:

- Cook and Holder [11] apply in their SUBDUE system the minimum description length (MDL) principle to discover substructures in a labeled data graph. The MDL principle states that the best pattern, is that pattern that minimizes the description length of the complete data graph. Hence, in SUBDUE a pattern is evaluated on how well it can compress the entire dataset. The input for the SUBDUE system is a labeled data graph; nodes and edges are labeled with non-unique labels. This is in contrast with the unique labels (‘constants’) in our system. But as we already noted, non-unique node labels and edge-labels can easily be simulated by constants, but the converse is not obvious. The SUBDUE system only mines patterns, no association rules.
- Ghazizadeh and Chawathe [16] mine in their SEuS system for connected subgraphs in a labeled, directed data graph, as in the SUBDUE system. Instead of generating candidate patterns using the input data graph, SEuS uses a summary of the data graph. This summary gives an upper bound for the support of the patterns, and the user can then select those patterns of which he wants to know the exact support. SEuS also only mines for frequent patterns and not for associations.
- Vanetik, Gudes, and Shimony [19] propose an Apriori-like [3] algorithm for mining subgraphs from a labeled data graph. The support of a graph pattern is defined as the maximal number of edge-disjoint instances of the pattern in the data graph. By reducing the support counting problem to the maximal independent set problem on graphs, they show that in worst case, computing the support of a graph pattern is NP-hard. They propose an Apriori-like algorithm to minimize the number of patterns for which the support needs to be computed. The major idea of their approach is using edge-disjoint paths as building blocks instead of items in classical itemset mining. Vanetik, Gudes, and Shimony also only mine for frequent patterns in the data graph.
- Kuramochi en Karypis [29] use the same support measure for graph patterns as Vanetik, Gudes and Shimony [19]. They also note that computing

the support of a graph pattern is NP-hard in worst case, since it can be reduced to finding the maximum independent set (MIS) in a graph. Kuramochi and Karypis quickly compute the support of a graph pattern using approximate MIS-algorithms. The number of candidate patterns is restricted using canonical labeling. As the majority of algorithms, Kuramochi and Karypis only mine for frequent patterns.

- Jeh and Widom [24] consider patterns that are, like our tree queries, inspired by conjunctive database queries, and they also emphasize the tree-shaped case. A severe restriction, however, is that their patterns can be matched by single nodes only, rather than by tuples of nodes. Still their work is interesting in that it presents a rather nonstandard approach to graph mining, quite different from the standard incremental, levelwise approach, and in that it incorporates ranking. Jeh and Widom mention association rules as an example of an application of their mining framework.

The related work that was most influential for us is Warmr [12, 13], although it belongs to the transactional category. Based on inductive logic programming, patterns in Warmr also feature existential variables and parameters. While not restricted to tree shapes, the queries in Warmr are restricted in another sense so that only transactional mining can be supported. Association rules in Warmr are defined in a naive manner through pattern extension, rather than being founded upon the theory of conjunctive query containment. The Warmr system is also Prolog-oriented, rather than database-oriented, which we believe is fundamental to mining of single large data graphs, and which allows a more uniform and parallel treatment of parameter instantiations, as we will show in this paper. Finally, Warmr does not seriously attempt to avoid the generation of duplicates. Yet, Warmr remains a pathbreaking work, which did not receive sufficient follow-up in the data mining community at large. We hope our present work represents an improvement in this respect. Many of the improvements we make to Warmr were already envisaged (but without concrete algorithms) in 2002 by Goethals and the second author [18].

Finally, we note that parameterized conjunctive database queries have been used in data mining quite early, e.g., [39, 38], but then in the setting of “data mining query languages”, where a *single* such query serves to specify a family of patterns to be mined or queried for, rather than the mining for such queries themselves, let alone associations among them.

3 Problem Statement

In this section we define some concepts formally. In the appendix an overview of all notations used in this paper is given.

We basically assume a set U of *data constants* from which the nodes of the data graph to be mined will be taken.

3.1 Graph-theoretic concepts

Let $N \subseteq U$ be any finite set of *nodes*; nodes can be any data objects such as numbers or strings. For our purposes, we define a (directed) *graph* on N as

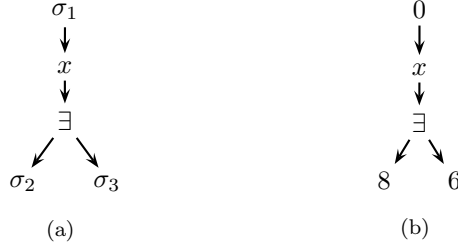


Figure 3: (a) is a parameterized tree pattern, and (b) is an instantiation of (a).

a subset of N^2 , i.e., as a finite set of ordered pairs of nodes. These pairs are called *edges*. We assume familiarity with the notion of a *tree* as a special kind of graph, and with standard graph-theoretic concepts such as *root* of a tree; *children*, *descendants*, *parent*, and *ancestors* of a node; and *path* in a graph. Any good algorithms textbook will supply the necessary background.

In this paper all trees we consider are rooted and unordered, unless stated otherwise.

3.2 Tree Pattern

Tree Patterns A *parameterized tree pattern* P is a tree whose nodes are called *variables*, and where additionally:

- Some variables may be marked as being *existential*;
- Some other variables may be marked as *parameters*;
- The variables of P that are neither existential nor parameters are called *distinguished*.

We will denote the set of existential variables by Π , the set of parameters by Σ , and the set of distinguished variables by Δ . To make clear that these sets belong to some parameterized tree pattern P we will use a subscript as in Π_P or Σ_P .

A *parameter assignment* α , for a parameterized tree pattern P , is a mapping $\Sigma \rightarrow U$ which assigns data constants to the parameters.

An *instantiated* tree pattern is a pair (P, α) , with P a parameterized tree pattern and α a parameter assignment for P . We will also denote this by P^α .

When depicting parameterized tree patterns, existential nodes are indicated by labeling them with the symbol ' \exists ' and parameters are indicated by labeling them with the symbol ' σ '. When depicting instantiated tree patterns, parameters are indicated by directly writing down their parameter assignment.

Figure 3 shows an illustration.

Matching Recall that a *homomorphism* from a graph G_1 to a graph G_2 is a mapping μ from the nodes of G_1 to the nodes of G_2 that preserves edges, i.e., if $(i, j) \in G_1$ then $(\mu(i), \mu(j)) \in G_2$. We now define a *matching* of an instantiated tree pattern P^α in a data graph G as a homomorphism μ from the underlying tree of P to G , with the constraint that for any parameter σ , if $\alpha(\sigma) = a$, then

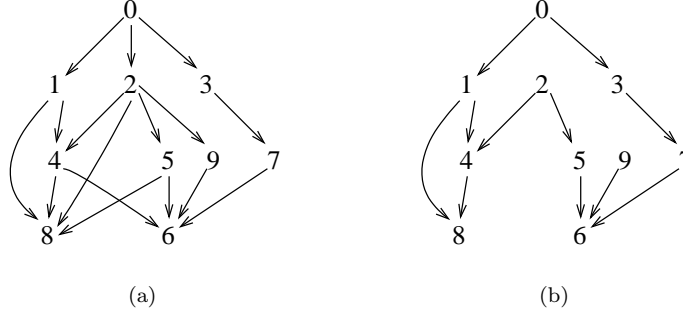


Figure 4: Two data graphs.

$\mu(\sigma)$ must be the node a . We denote the set $\{\mu|_{\Delta} : \mu \text{ is a matching of } P^{\alpha} \text{ in } G\}$ by $P^{\alpha}(G)$.

Frequency of a tree pattern The *frequency* of an instantiated tree pattern P^{α} in a data graph G , is formally defined as the cardinality of $P^{\alpha}(G)$. So, we count the number of matchings of P^{α} in G , with the important provision that *we identify any two matchings that agree on the distinguished variables*. Indeed, two matchings that differ only on the existential nodes need not be distinguished, as this is precisely the intended semantics of existential nodes. Note that we do not need to worry about selected nodes, as all matchings will agree on those by definition. For a given threshold k (a natural number) we say that P^{α} is *k-frequent* if its frequency is at least k . Often the threshold is understood implicitly, and then we talk simply about “frequent” patterns and denote the threshold by *minsup*.

Example. Take again the instantiated tree pattern P^{α} shown in Figure 3(b). Let us name the existential node by y ; let us name the parameter labeled 0 by z_1 ; the parameter labeled 8 by z_2 ; and the parameter labeled 6 by z_3 . The distinguished node already has the name x . Now let us apply P^{α} to the simple example data graph G shown in Figure 4(a). The following table lists all matchings of P^{α} in G :

	z_1	x	y	z_2	z_3
h_1	0	1	4	8	6
h_2	0	2	4	8	6
h_3	0	2	5	8	6

As required by the definition, all matchings match z_1 to 0, z_2 to 8, and z_3 to 6. Although there are three matchings, when determining the frequency of P^{α} in G , we only look at their value on x to distinguish them, as y is existential. So, h_2 and h_3 are identified as identical matchings when counting the number of matchings. In conclusion, the frequency of P^{α} in G is two, as x can be matched to the two different nodes 1 and 2. \square

3.3 Tree Query

Tree Queries A *parameterized tree query* Q is a pair (H, P) where:

1. P is a parameterized tree pattern, called the *body* of Q ;

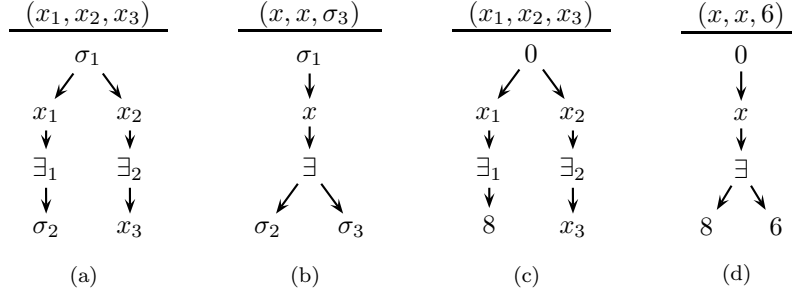


Figure 5: (a) and (b) are parameterized tree queries; (c) is an instantiation of (a); (d) is an instantiation of (b); and query (b) is ρ -contained in query (a)

2. H is a tuple of distinguished variables and parameters coming from P . All distinguished variables of P must appear at least once in H . We call H the *head* of Q .

A parameter assignment for Q is simply a parameter assignment for its body, and an *instantiated* tree query is then again a pair (Q, α) with Q a parameterized tree query and α a parameter assignment for Q . We will again also denote this by Q^α .

When depicting tree queries, the head is given above a horizontal line, and the body below it. Two illustrations are given in Figure 5.

Frequency of a tree query The *frequency* of an instantiated tree query $Q^\alpha = ((H, P), \alpha)$ in a data graph G , is defined as the frequency of the body P^α in G . When G is understood, we denote the frequency by $\text{Freq}(P^\alpha)$. For a given threshold k (a natural number) we say that Q^α is *k-frequent* if its frequency is at least k . Again, this threshold is often understood implicitly, and then we talk simply about “frequent” queries and denote the threshold by *minsup*.

Containment of tree queries An important step towards our formal definition of tree-query association is the notion of *containment* among queries. Since queries are parameterized, a variation of the classical notion of containment [9, 40, 2] is needed in that we now need to specify a parameter correspondence.

First, we define the *answer set* of an instantiated tree query Q^α , with $Q = (H, P)$, in a data graph G as follows:

$$Q^\alpha(G) := \{\mu(H) \mid \mu \text{ is a matching of } P^\alpha \text{ in } G\}$$

Consider two parameterized tree queries Q_1 and Q_2 , with $Q_i = (H_i, P_i)$ for $i = 1, 2$. A *parameter correspondence* from Q_1 to Q_2 is any mapping $\rho : \Sigma_1 \rightarrow \Sigma_2$. We then say that a parameterized tree query Q_2 is ρ -*contained* in a parameterized tree query Q_1 , if for every α_2 , a parameter assignment for Q_2 , $Q_2^{\alpha_2}(G) \subseteq Q_1^{\alpha_2 \circ \rho}(G)$ for all data graphs G . In shorthand notation we write this as $Q_2 \subseteq_\rho Q_1$.

Containment as just defined is a semantical property, referring to all possible data graphs, and it is not immediately clear how one could decide this property syntactically. The required syntactical notion for this is that of ρ -*containment*

mapping, which we next define in two steps. For the tree queries Q_1 and Q_2 as above, and ρ a parameter correspondence from Q_1 to Q_2 :

1. A ρ -containment mapping from P_1 to P_2 is a homomorphism f from the underlying tree of P_1 to the underlying tree of P_2 , with the properties:
 - (a) f maps the distinguished nodes of P_1 to distinguished nodes or parameters of P_2 ; and
 - (b) $f|_{\Sigma_1} = \rho$, i.e., for each $z \in \Sigma_1$ we have $f(z) = \rho(z)$.
2. Finally, a ρ -containment mapping from Q_1 to Q_2 is a ρ -containment mapping f from P_1 to P_2 such that $f(H_1) = H_2$.

For later use, we note:

Lemma 1. *Consider three parameterized tree patterns P_1 , P_2 , and P_3 , a parameter correspondence $\rho_1 : \Sigma_1 \rightarrow \Sigma_2$, a parameter correspondence $\rho_2 : \Sigma_2 \rightarrow \Sigma_3$, a ρ_1 -containment mapping f_1 from P_1 to P_2 , and a ρ_2 -containment mapping f_2 from P_2 to P_3 . Then $f_2 \circ f_1$ is a $(\rho_2 \circ \rho_1)$ -containment mapping from P_1 to P_3 .*

Proof. We will show that:

1. $f_2 \circ f_1$ is homomorphism;
2. $f_2 \circ f_1$ maps distinguished nodes of P_1 to distinguished nodes or parameters of P_3 ; and
3. $(f_2 \circ f_1)|_{\Sigma_1} = \rho_2 \circ \rho_1$.

(1) Clearly $f_2 \circ f_1$ is a homomorphism since both f_1 and f_2 are homomorphisms, and it is already known that a composition of homomorphisms is a homomorphism.

(2) Consider a $x_1 \in \Delta_1$, then there are two possibilities for $f_1(x_1)$:

1. $f_1(x_1) = x_2$, with $x_2 \in \Delta_2$. Then we know, since f_2 is a ρ_2 -containment mapping, that $f_2(x_2)$ is either a distinguished node $x_3 \in \Delta_3$, or a parameter $z_3 \in \Sigma_3$.
2. $f_1(x_1) = z_2$, with $z_2 \in \Sigma_2$. Then we know, since $f_2|_{\Sigma_2} = \rho_2$, that $f_2(z_2) = z_3$, with $z_3 \in \Sigma_3$.

Hence, we can conclude that $f_2 \circ f_1$ maps distinguished nodes of P_1 to distinguished nodes or parameters of P_3 .

(3) For each $z_1 \in \Sigma_1$, we have $f_2(f_1(z_1)) = \rho_2(\rho_1(z_1))$. Hence, $(f_2 \circ f_1)|_{\Sigma_1} = \rho_2 \circ \rho_1$. \square

From the theory of conjunctive database queries [9, 40, 2] we can derive the following:

Lemma 2. *Consider two parameterized tree queries Q_1 and Q_2 , with $Q_1 = (H_1, P_1)$ and $Q_2 = (H_2, P_2)$ and a parameter correspondence $\rho : \Sigma_1 \rightarrow \Sigma_2$. Then Q_2 is ρ -contained in Q_1 ($Q_2 \subseteq_\rho Q_1$), if and only if there exists a ρ -containment mapping from Q_1 to Q_2 .*

Proof. Let us start with the ‘only if’ direction. We first introduce the concept of a *freezing* of a parameterized tree query $Q = (H, P)$. Recall that U is the set of data constants from which the nodes of the data graph to be mined will be taken. A freezing β of P is then a one-to-one mapping from the nodes of P to U . We denote by $\text{freeze}_\beta(P)$ the data graph constructed from P by replacing each node n of P by $\beta(n)$, and we denote by $\text{freeze}_\beta(H)$ the tuple constructed from H by replacing each node n in H by the data constant $\beta(n)$.

For example, consider the parameterized tree query $Q = (H, P)$ in Figure 6(a). Figure 6(b), shows $\text{freeze}_\beta(P)$ and $\text{freeze}_\beta(H)$ for the freezing β given as follows: $x_1 \rightarrow c_1$; $x_2 \rightarrow c_2$; $\exists_3 \rightarrow c_3$; $x_4 \rightarrow c_4$; $x_5 \rightarrow c_5$; $\sigma_6 \rightarrow c_6$.

We can now continue with the proof of the ‘only if’ direction. Consider a freezing β from the nodes of P_2 to U . Note that $\beta|_{\Sigma_2}$ is a parameter assignment for Q_2 , and $\text{freeze}_\beta(H_2) \in Q_2^{\beta|_{\Sigma_2}}$ ($\text{freeze}(P_2)$). Since $Q_2 \subseteq_\rho Q_1$, also $\text{freeze}_\beta(H_2) \in Q_1^{\beta|_{\Sigma_2} \circ \rho}$ ($\text{freeze}(P_2)$). Hence, there must be a matching μ from $P_1^{\beta|_{\Sigma_2} \circ \rho}$ in $\text{freeze}_\beta(P_2)$ such that $\mu(H_1) = \text{freeze}_\beta(H_2)$. Now consider the function $g : \beta^{-1} \circ \mu$. We show that g is ρ -containment mapping from Q_1 to Q_2 :

1. Clearly, g is a homomorphism from P_1 to P_2 since μ is a homomorphism and β^{-1} is an isomorphism. Also the following properties hold for g :
 - (a) g maps distinguished nodes of P_1 to distinguished nodes or parameters of P_2 since $g(H_1) = H_2$ (as shown in (2)); and
 - (b) for each $z \in \Sigma_1$: $g(z) = \beta^{-1}(\mu(z)) = \beta^{-1}(\beta(\rho(z))) = \rho(z)$, hence $g|_{\Sigma_1} = \rho$
2. $g(H_1) = \beta^{-1}(\mu(H_1)) = \beta^{-1}(\text{freeze}_\beta(H_2)) = H_2$.

Hence, we conclude that g is a ρ -containment mapping from Q_1 to Q_2 .

Let us then look at the ‘if’ direction. Let h be the ρ -containment mapping from Q_1 to Q_2 . Consider an arbitrary parameter assignment α_2 for Q_2 . We must prove that for every data graph G , if $a \in Q_2^{\alpha_2}(G)$, then also $a \in Q_1^{\alpha_1 \circ \rho}(G)$. Consider such an arbitrary data graph G . Since, $a \in Q_2^{\alpha_2}(G)$, we know that there exists a matching μ of $P_2^{\alpha_2}$ in G such that $a = \mu(H_2)$. Now consider the function $g = \mu \circ h$. We show that g is a matching from $P_1^{\alpha_1 \circ \rho}$ in G and $a = g(H_1)$:

1. g is a homomorphism since both μ and h are homomorphisms; and
2. for each $z \in \Sigma_1$ we have $g(z) = \mu(h(z)) = \mu(\rho(z)) = \alpha_2(\rho(z))$.

So, g is indeed a matching of $P_1^{\alpha_1 \circ \rho}$ in G . Finally, we observe that $g(H_1) = \mu(h(H_1)) = \mu(H_2) = a$, as desired. \square

Checking for a containment mapping is evidently computable, and although the problem for general database conjunctive queries is NP-complete, our restriction to tree shapes allows for efficient checking, as we will see later.

Example. Consider the parameterized and instantiated tree queries shown in Figure 5. In the example data graph in Figure 4(a) the frequency of query (c) is 10 and that of query (d) is 2. Let Σ_a be the set of parameters of query (a), and let Σ_b be the set of parameters of query (b); then let the parameter correspondence $\rho : \Sigma_a \rightarrow \Sigma_b$ be as follows: $\sigma_1 \rightarrow \sigma_1$; $\sigma_2 \rightarrow \sigma_2$. A moment's

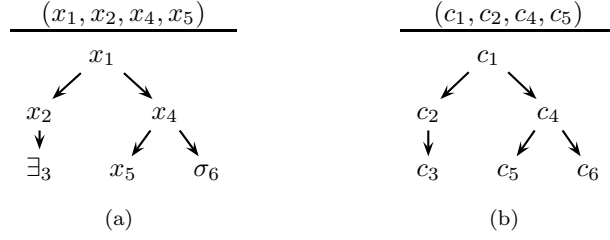


Figure 6: (b) is a freezing of the parameterized tree query in (a)

reflection should convince the reader that (b) is ρ -contained in (a), and indeed a ρ -containment mapping f from (a) to (b) can be found as follows:

f	
σ_1	σ_1
x_1	x
x_2	x
\exists_1	\exists
\exists_2	\exists
σ_2	σ_2
x_3	σ_3

3.4 Tree-Query Association

Association Rules A *parameterized association rule* (pAR) is of the form $Q_1 \Rightarrow_\rho Q_2$, with Q_1 and Q_2 parameterized tree queries and ρ a parameter correspondence from Σ_1 to Σ_2 . We call a pAR *legal* if $Q_2 \subseteq_\rho Q_1$. We call Q_1 the left-hand side (lhs), and Q_2 the right-hand side (rhs). A *parameter assignment* α , for a pAR, is a mapping $\Sigma_2 \rightarrow U$ which assigns data constants to the parameters. An *instantiated association rule* (iAR) is a pair $(Q_1 \Rightarrow_\rho Q_2, \alpha)$, with $Q_1 \Rightarrow_\rho Q_2$ a pAR and α a parameter assignment for $Q_1 \Rightarrow_\rho Q_2$. Note that while α is only defined on the rhs, we can also apply it to the lhs by using ρ first.

Confidence The *confidence* of an iAR in a data graph G is defined as the frequency of $Q_2^{\alpha_2}$ divided by the frequency of $Q_1^{\alpha_2 \circ \rho}$. If the AR is legal, we know that the answer set of $Q_2^{\alpha_2}$ is a subset of the answer set of $Q_1^{\alpha_2 \circ \rho}$, and hence the confidence equals precisely the proportion that the $Q_2^{\alpha_2}$ answer set takes up in the $Q_1^{\alpha_2 \circ \rho}$ answer set. Thus, our notions of a legal pAR and confidence are very intuitive and natural.

For a given threshold c (a rational number, $0 \leq c \leq 1$) we say that the iAR is *c-confident* in G if its confidence in G is at least c . Often the threshold is understood implicitly, and then we talk simply about “confident” iARs and denote the threshold by *minconf*.

Furthermore, the iAR is called *frequent* in G if $Q_2^{\alpha_2}$ is frequent in G . Note that if the iAR is legal and frequent, then also $Q_1^{\alpha_2 \circ \rho}$ is frequent, since the rhs is ρ -contained in the lhs.

Example. Continuing the previous example, we can see that we can form a legal pAR from the queries of Figure 5, with (a) the lhs and (b) the rhs and ρ as follows: $\sigma_1 \rightarrow \sigma_1$; $\sigma_2 \rightarrow \sigma_2$. We can also form an iAR with the tree queries in Figure 5(c) and Figure 5(d); the confidence of this iAR in the data graph of Figure 4(a) is 2/10. Many more examples of ARs are given in Section 5.

3.5 Mining Problems

We are now finally ready to define the graph mining problems we want to solve.

3.5.1 Mining Tree Queries

Input: A data graph G ; a threshold *minsup*.

Output: All frequent instantiated tree queries $Q = ((H, P), \alpha)$.

In theory, however, there are infinitely many k -frequent tree queries, and even if we set an upper bound on the size of the patterns, there may be exponentially many. As an extreme example, if G is the complete graph on the set of nodes $\{1, \dots, n\}$, and $k \leq n$, then *any* instantiated pattern with all parameters assigned to values in $\{1, \dots, n\}$, and with at least one distinguished variable, is frequent.

Hence, in practice, we want an algorithm that runs incrementally, and that can be stopped any time it has run long enough or has produced enough results. We introduce such an algorithm in Section 4.

3.5.2 Association Rule Mining

Input: A data graph G ; a threshold *minsup*; a parameterized tree query Q_{left} ; and a threshold *minconf*.

Output: All iARs $(Q_{\text{left}} \Rightarrow_{\rho} Q_{\text{right}}, \alpha)$ that are legal, frequent and confident in G

In theory, however, there are infinitely many legal, frequent and confident association rules for a fixed lhs, and even if we set an upper bound on the size of the rhs, there may be exponentially many. Hence, in practice, we want an algorithm that runs incrementally, and that can be stopped any time it has run long enough or has produced enough results. We introduce such an algorithm in Section 5.

4 Mining Tree Queries

In this Section we present an algorithm for mining frequent instantiated tree queries in a large data graph. But first we show that we do not need to tackle the problem in its full generality.

4.1 Problem Reduction

In this subsection we show that, without loss of generality, we can focus on parameterized tree queries that are ‘pure’.

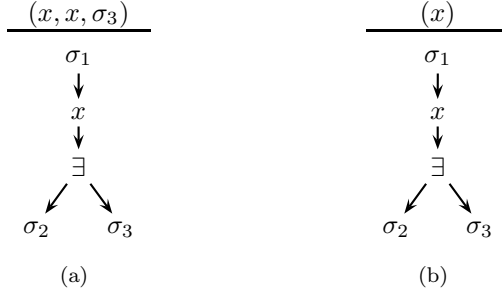


Figure 7: (a) is an impure parameterized tree query. The parameterized tree query in (b) is the purification of the parameterized tree query (a), and expresses precisely the same information.

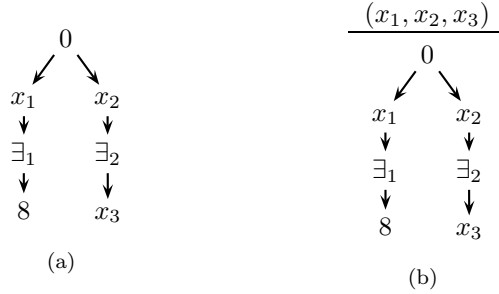


Figure 8: (b) is the pure instantiated tree query constructed from the instantiated tree pattern in (a)

Pure Tree Queries To define this formally, assume that all possible variables (nodes of tree patterns) have been arranged in some fixed but arbitrary order. We then call a parameterized tree query $Q = (H, P)$ *pure* when H consists of the enumeration, in order and without repetitions, of all the distinguished variables of P . In particular H cannot contain parameters. We call H the *pure head* for P . As an illustration, the parameterized tree query in Figure 5(a) is pure, while the parameterized tree query in Figure 5(b) is not pure.

A parameterized tree query that is not pure can always be rewritten to a parameterized tree query that is pure, in such a way that all instantiations of the impure query correspond to instantiations of the pure query, with the same frequency. Indeed, take a parameterized tree query $Q = (H, P)$. We can purify Q by removing all parameters and repetitions of distinguished variables from H , and sort H by the order on the variables. An illustration of this is given in Figure 7.

We can conclude that it is sufficient to only consider pure instantiated tree queries. As a consequence, rather than mining tree *queries*, it suffices to mine for tree *patterns*, because the frequency of a query is nothing else then the frequency of his body, i.e., a pattern. An illustration is given in Figure 8.

4.2 Overall Approach

An overall outline of our tree-query mining algorithm is the following:

Outer loop: Generate, incrementally, all possible trees T of increasing sizes. Avoid trees that are isomorphic to previously generated ones.

Inner loop: For each T , generate all instantiated tree patterns P^α based on T , and test their frequency.

The algorithm is incremental in the number of nodes of the pattern. We generate canonically ordered rooted trees of increasing sizes, avoiding the generation of isomorphic duplicates. It is well known how to do this efficiently [37, 30, 42, 10]. Note that this generation of trees is in no way “levelwise” [31]. Indeed, under the way we count pattern occurrences, a subgraph of a pattern might be less frequent than the pattern itself (this was already pointed out by Kuramochi and Karypis [29]). So, our algorithm systematically considers ever larger trees, and can be stopped any time it has run long enough or has produced enough results. Our algorithm does not need any space beyond what is needed to store the mining results. The outer loop of our algorithm will be explained in more detail in Section 4.3.

For each tree, all conjunctive queries based on that tree are generated. Here, we do work in a levelwise fashion. This aspect of our algorithm has clear similarities with “query flocks” [39]. A query flock is a user-specified conjunctive query, in which some constants are left unspecified and viewed as parameters. A levelwise algorithm was proposed for mining all instantiations of the parameters under which the resulting query returns enough answers. We push that approach further by also mining the query flocks themselves. Consequently, the specialization relation on queries used to guide the levelwise search is quite different in our approach. The inner loop of our algorithm will be explained in more detail in Section 4.4.

A query based on some tree may be equivalent to a query based on a previously seen tree. Furthermore, two queries based on the same tree may be equivalent. We carefully and efficiently avoid the counting of equivalent queries, by using and adapting what is known from the theory of conjunctive database queries. This will be discussed in Section 4.5.

4.3 Outer Loop

In the outer loop we generate all possible trees of increasing sizes and we avoid trees that are isomorphic to previously generated ones. In fact, it is well known how to do this [37, 30, 42, 10]. What these procedures typically do is generating trees that are *canonically ordered* in the following sense. Given an (unordered) tree T , we can order the children of every node in some way, and call this an *ordering* of T . For example, Figure 9 shows two orderings of the same tree. From the different orderings of a tree T , we want to uniquely select one, to be the canonical ordering of T . For each such possible ordering of T , we can write down the *level sequence* of the resulting tree. This is actually a string representation of the resulting tree. This level sequence is as follows: if the tree has n nodes then this is a sequence of n numbers, where the i th number is the depth of the i th node in preorder. Here, the depth of the root is 0, the depth of



Figure 9: Two orderings of the same tree. The left one is canonical.

its children is 1, and so on. The canonical ordering of T is then the ordering of T that yields the lexicographically maximal level sequence among all possible orderings of T .

For example, in Figure 9, the left one is the canonical one.

4.4 Inner Loop

Let G be the data graph being mined, and let U be its set of nodes. In this section, we fix a tree T , and we want to find all instantiated tree patterns P^α based on T whose frequency in G is at least *minsup*.

This task lends itself naturally to a levelwise approach [31]. A natural choice for the specialization relation is suggested by an alternative notation for the patterns under consideration. Concretely, since the underlying tree T is fixed, any parameterized tree pattern P based on T is characterized by two parameters:

1. The set Π of existential nodes;
2. The set Σ of parameters.

Note that Π and Σ are disjoint.

Thus, a parameterized tree pattern P is completely characterized by the pair (Π, Σ) . An instantiation P^α of P is then represented by the triple (Π, Σ, α) . For two parameterized tree patterns $P_1 = (\Pi_1, \Sigma_1)$ and $P_2 = (\Pi_2, \Sigma_2)$ we now say that P_1 *specializes* P_2 if $\Pi_1 \supseteq \Pi_2$ and $\Sigma_1 \supseteq \Sigma_2$; and $\alpha_2 = \alpha_1|_{\Sigma_2}$. We also say that P_2 *generalizes* P_1 .

Parent An immediate generalization of a tree pattern is called a *parent*. Formally, let $P = (\Pi, \Sigma)$ and $P' = (\Pi', \Sigma')$ be parameterized tree patterns based on T . We say that P' is a *parent* of P if:

- (i) $\Sigma = \Sigma'$ and $\Pi = \Pi' \cup \{y\}$ for some node $y \notin \Pi'$; or
- (ii) $\Pi = \Pi'$ and $\Sigma = \Sigma' \cup \{z\}$ for some node $z \notin \Sigma'$.

From the following lemma, it follows that specialized patterns have a lower frequency, as expected for a specialization relation:

Lemma 3. *Let P and P' be parameterized tree patterns such that P' is a parent of P . Let P^α be an instantiation of P , and let $\alpha' = \alpha|_{\Sigma'}$. Then $\text{Freq}(P^\alpha) \leq \text{Freq}(P'^{\alpha'})$.*

Proof. We will show that $\#P^\alpha(G) \leq \#P'^{\alpha'}(G)$ by defining an injection $I : P^\alpha(G) \rightarrow P'^{\alpha'}(G)$.

Since P' is a parent of P , we know that $\Delta' = \Delta \cup \{u\}$ where u is either an existential node or a parameter of P . Note that each $\mu \in P^\alpha(G)$ is of the form $\bar{\mu}|_\Delta$ for some matching $\bar{\mu}$ of $P^\alpha \in G$. For each μ in $P^\alpha(G)$, we fix arbitrarily $\bar{\mu}$. Now we define $I(\mu) := \bar{\mu}|_{\Delta'}$. To see that I is an injection, let $\mu_1, \mu_2 \in P^\alpha(G)$ and suppose that $I(\mu_1) = I(\mu_2)$. In other words, $\bar{\mu}_1|_{\Delta'} = \bar{\mu}_2|_{\Delta'}$. In particular, $\mu_1 = \bar{\mu}_1|_\Delta = \bar{\mu}_2|_\Delta = \mu_2$, as desired.

Hence, we can conclude that $\#P^\alpha(G) \leq \#P'^{\alpha'}(G)$ and that $\text{Freq}(P^\alpha) \leq \text{Freq}(P'^{\alpha'})$. \square

The above lemma suggests the following definition of specialization among *instantiated* tree patterns: we say that $(\Pi_1, \Sigma_1, \alpha_1)$ is a specialization of $(\Pi_2, \Sigma_2, \alpha_2)$ if the parameterized tree pattern (Π_1, Σ_1) is a specialization of the parameterized tree pattern (Π_2, Σ_2) , and $\alpha_2 = \alpha_1|_{\Sigma_2}$.

Intuitively, the previous lemma then expresses that the frequency of an instantiated tree pattern is always at most the frequency of any of its instantiated parents.

4.4.1 Candidate generation

Candidate pattern A *candidate pattern* is an instantiated tree pattern whose frequency is not yet determined, but all whose generalizations are known to be frequent.

Using the specialization relation and the definition for a candidate pattern we explain how the levelwise search for frequent instantiated tree patterns will go.

Levelwise search We start with the most general instantiated tree pattern $P = (\emptyset, \emptyset, \emptyset)$, and we progressively consider more specific patterns. The search has the typical property that, in each new iteration, new *candidate* patterns are generated; the frequency of all newly discovered candidate patterns is determined, and the process repeats.

There are many different instantiations to consider for each parameterized tree pattern. Hence, to generate candidate patterns in an efficient manner, we propose the use of *candidacy tables* and *frequency tables*. These candidacy and frequency tables allow us to generate all frequent instantiations for a particular parameterized tree pattern in parallel. A frequency table contains all frequent instantiations for a particular parameterized tree pattern.

Formally, for any parameterized pattern $P = (\Pi, \Sigma)$, we define:

$$\begin{aligned} \text{CanTab}_{\Pi, \Sigma} &= \{\alpha \mid P^\alpha \text{ is a candidate instantiated tree pattern}\} \\ \text{FreqTab}_{\Pi, \Sigma} &= \{\alpha \mid P^\alpha \text{ is a frequent instantiated tree pattern}\} \end{aligned}$$

Technically, the table has columns for the different parameters, plus a column **freq**. Note that when $\Sigma = \emptyset$, i.e., P has no parameters, this is a single-column, single-row table containing just the frequency of P . This still makes sense and can be interpreted as boolean values; for example, if $\text{FreqTab}_{\Pi, \emptyset}$ contains the empty tuple, then the pattern $(\Pi, \emptyset, \emptyset)$ is frequent; if the table is empty, the pattern is not frequent. Of course in practice, all frequency tables for parameterless patterns can be combined into a single table. All frequency tables are kept in a relational database.

The following crucial lemma shows these tables can be populated efficiently.

Join Lemma. *A parameter assignment α is in $CanTab_{\Pi, \Sigma}$ if and only if the following conditions are satisfied for every parent (Π', Σ') of (Π, Σ) :*

- (i) *If $\Pi = \Pi'$, then $\alpha|_{\Sigma'} \in FreqTab_{\Pi', \Sigma'}$;*
- (ii) *If $\Sigma = \Sigma'$, then $\alpha \in FreqTab_{\Pi', \Sigma'}$.*

Proof. For the ‘only-if’ direction: By definition of a candidacy table, if $\alpha \in CanTab_{\Pi, \Sigma}$, then all generalizations of (Π, Σ, α) are frequent. In particular, for all parents (Π', Σ') of (Π, Σ) , we know that $(\Pi', \Sigma', \alpha|_{\Sigma'})$ is frequent, since parents are generalizations.

For the ‘if’ direction, we must show that all generalizations of (Π, Σ, α) are frequent. Consider such a generalization $(\Pi_{g_1}, \Sigma_{g_1}, \alpha|_{\Sigma_{g_1}})$. Let us denote the parent relation by \geq_p . Then there is a sequence of parent patterns: $(\Pi_{g_1}, \Sigma_{g_1}) \geq_p (\Pi_{g_2}, \Sigma_{g_2}) \geq_p \dots \geq_p (\Pi', \Sigma')$. And we have: $Freq(\Pi_{g_1}, \Sigma_{g_1}, \alpha|_{\Sigma_{g_1}}) \geq Freq(\Pi_{g_2}, \Sigma_{g_2}, \alpha|_{\Sigma_{g_2}}) \geq \dots \geq Freq(\Pi', \Sigma', \alpha|_{\Sigma'}) \geq \text{minsup}$. The last inequality is given by (i) or (ii), the other inequalities are given by Lemma 3. \square

The Join Lemma has its name because, viewing the tables as relational database tables, it can be phrased as follows:

Each candidacy table can be computed by taking the natural join of its parent frequency tables.

The only exception is when $\Pi = \emptyset$ and $\Sigma = \{z\}$ is a singleton; this is the initial iteration of the search process, when there are no constants in the parent tables to start from. In that case, we define $CanTab_{\emptyset, \{z\}}$ as the table with a single column z , holding all nodes of the data graph G being mined.

4.4.2 Frequency counting using SQL

The search process starts by determining the frequency of the underlying tree $T = (\emptyset, \emptyset)$; indeed, formally this amounts to computing $FreqTab_{\emptyset, \emptyset}$. Similarly, for each parameterized tree pattern $P = (\Pi, \emptyset)$ with $\Pi \neq \emptyset$, all we can do is determine its frequency, except that here, we do this only on condition that its parent patterns are frequent.

We have seen above that, if the frequency tables are viewed as relational database tables, we can compute each candidacy table by a single database query, using the Join Lemma. Now suppose the data graph G that is being mined is stored in the relational database system as well, in the form of a table $G(\text{from}, \text{to})$. Then also each frequency table can be computed by a single SQL query.

Indeed, in the cases where $\Sigma = \emptyset$ this simply amounts to formulating the pattern in SQL, and determining its count (eliminating duplicates). Since our patterns are in fact conjunctive queries (or datalog rules) known from database research [2, 40]. They can easily be translated in SQL:

- The FROM-clause consists of all table references of the form G as G_{ij} , for all edges $x_i \rightarrow x_j$ in T .
- The WHERE-clause consists of all equalities of the form $G_{ij}.\text{from} = G_{ik}.\text{from}$ as well of equalities of the form $G_{ij}.\text{to} = G_{jh}.\text{from}$.

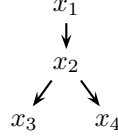


Figure 10: Illustration on translating a tree pattern without parameters in SQL.

- The **SELECT**-clause is of the form **SELECT DISTINCT** and consists of all column references of the form **G_{ij}.to** when x_{ij} is a distinguished node in P , plus one reference of the form **G_{1k}.from** if the root node is distinguished.

The SQL query for the tree in Figure 10 with $\Pi = \{x_2\}$ and $\Sigma = \emptyset$ is as follows:

```

E = SELECT G12.from, G23.to, G24.to
FROM G as G12, G as G23, G as G24
WHERE G12.to = G23.from AND G12.to = G24.from

```

But also when $\Sigma \neq \emptyset$, we can compute $FreqTab_{\Pi, \Sigma}$ by a single SQL query. Note that we thus compute the frequency of a large number of instantiated tree patterns in parallel! We proceed as follows:

1. we formulate the pattern (Π, \emptyset) in SQL; call the resulting expression E
2. We then take the natural join of E and $CanTab_{\Pi, \Sigma}$, group by Σ , and count each group.

The join with the candidacy table ensures that only candidate patterns are counted.

For instance, the SQL query to compute the frequency table for the tree in Figure 10, with $\Pi = \{x_2\}$ and $\Sigma = \{x_1, x_3\}$, with E as above, is as follows:

```

SELECT E.x1, E.x3, COUNT(*)
FROM E, CanTab_{x2}, {x1, x3} CT
WHERE E.x1 = CT.x1 AND E.x3 = CT.x3
GROUP BY E.x1, E.x3 HAVING COUNT(*) >= minsup

```

It goes without saying that, whenever the frequency table of a tree pattern is found to be empty, the search for more specialized patterns is pruned at that point.

4.4.3 The algorithm

Putting everything together so far, the algorithm is given in Algorithm 1. In outline it is a double Apriori algorithm [3], where the sets Π form one dimension of itemsets, and the sets Σ another. A graphical illustration of the algorithm is given in Figure 11. In this illustration we use *tries* (or prefix-trees) to store the itemsets. A trie [5, 7, 26] is commonly used in implementations of the Apriori algorithm.

Algorithm 1 Levelwise search for frequent tree patterns.

```

1: for each unordered, rooted tree  $T$  do
2:    $X :=$  set of nodes of  $T$ 
3:    $p := 0$ ;  $\mathcal{P}_0 := \{\emptyset\}$ 
4:   repeat
5:     for each  $\Pi \in \mathcal{P}_p$  do
6:       Compute  $FreqTab_{\Pi, \emptyset}$  in SQL
7:       if  $FreqTab_{\Pi, \emptyset} \neq \emptyset$  then
8:          $s := 1$ 
9:          $\mathcal{S}_1 := \{\{z\} \mid z \in X - \Pi\}$ 
10:        repeat
11:          for each  $\Sigma \in \mathcal{S}_s$  do
12:            if  $p = 0$  and  $s = 1$  then
13:               $CanTab_{\Pi, \Sigma} :=$  set of nodes of  $G$ 
14:            else
15:               $CanTab_{\Pi, \Sigma} := \bowtie \{FreqTab_{\Pi', \Sigma'} \mid (\Pi', \Sigma') \text{ parent of } (\Pi, \Sigma)\}$ 
16:            end if
17:            Compute  $FreqTab_{\Pi, \Sigma}$  in SQL
18:            if  $FreqTab_{\Pi, \Sigma} = \emptyset$  then
19:              remove  $\Sigma$  from  $\mathcal{S}_s$   $\{\Sigma \text{ is pruned away}\}$ 
20:            end if
21:          end for
22:           $\mathcal{S}_{s+1} := \{\Sigma \subseteq X - \Pi \mid \#\Sigma = s + 1$ 
23:                    and each  $s$ -subset of  $\Sigma$  is in  $\mathcal{S}_s\}$ 
24:           $s := s + 1$ 
25:        until  $\mathcal{S}_s = \emptyset$ 
26:      else
27:        remove  $\Pi$  from  $\mathcal{P}_p$   $\{\Pi \text{ is pruned away}\}$ 
28:      end if
29:    end for
30:     $\mathcal{P}_{p+1} := \{\Pi \subseteq X \mid \#\Pi = p + 1 \text{ and each } p\text{-subset of } \Pi \text{ is in } \mathcal{P}_p\}$ 
31:     $p := p + 1$ 
32:  until  $\mathcal{P}_p = \emptyset$ 
33: end for

```

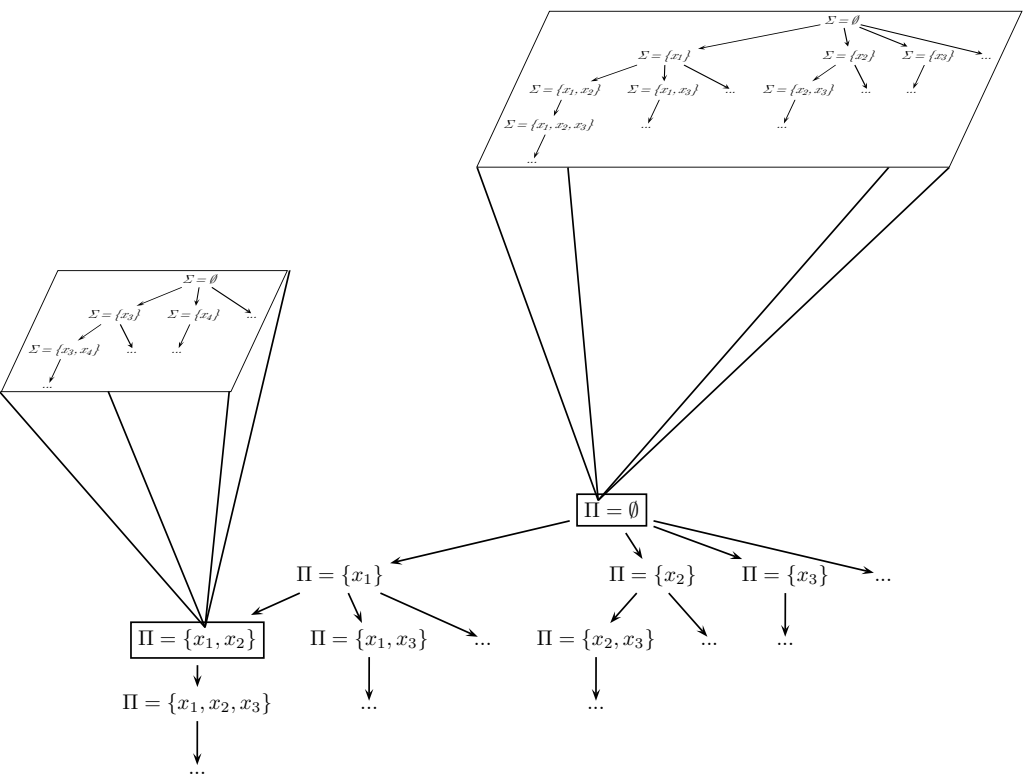


Figure 11: Illustration of the algorithm in Algorithm 1

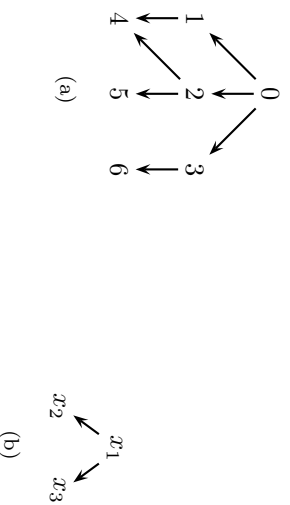
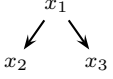
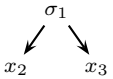

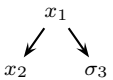
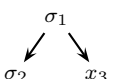
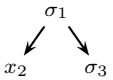
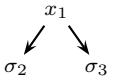
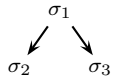
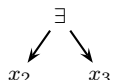
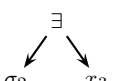
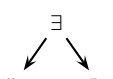


Figure 12: Data graph G and unordered rooted tree T for the example run in Section 4.4.4.

4.4.4 Example run

In this Section we give an example run of the proposed algorithm in Algorithm 1. Consider the example data graph G in Figure 12(a); the unordered rooted tree T in Figure 12(b); and let the minimum support threshold be 3.

The example run then looks as follows:

Π	Σ	P	$CanTab$	$FreqTab$												
\emptyset	\emptyset			<table><tr><th>$Freq$</th></tr><tr><td>15</td></tr></table>	$Freq$	15										
$Freq$																
15																
\emptyset	$\{x_1\}$		nodes of G	<table><tr><th>σ_1</th><th>$Freq$</th></tr><tr><td>0</td><td>9</td></tr><tr><td>2</td><td>4</td></tr></table>	σ_1	$Freq$	0	9	2	4						
σ_1	$Freq$															
0	9															
2	4															
\emptyset	$\{x_2\}$		nodes of G	<table><tr><th>σ_2</th><th>$Freq$</th></tr><tr><td>1</td><td>3</td></tr><tr><td>2</td><td>3</td></tr><tr><td>3</td><td>3</td></tr><tr><td>4</td><td>3</td></tr></table>	σ_2	$Freq$	1	3	2	3	3	3	4	3		
σ_2	$Freq$															
1	3															
2	3															
3	3															
4	3															
\emptyset	$\{x_3\}$		nodes of G	<table><tr><th>σ_3</th><th>$Freq$</th></tr><tr><td>1</td><td>3</td></tr><tr><td>2</td><td>3</td></tr><tr><td>3</td><td>3</td></tr><tr><td>4</td><td>3</td></tr></table>	σ_3	$Freq$	1	3	2	3	3	3	4	3		
σ_3	$Freq$															
1	3															
2	3															
3	3															
4	3															
\emptyset	$\{x_1, x_2\}$		$FreqTab_{\emptyset, \{x_1\}}$ $\bowtie FreqTab_{\emptyset, \{x_2\}}$	<table><tr><th>σ_1</th><th>σ_2</th><th>$Freq$</th></tr><tr><td>0</td><td>1</td><td>3</td></tr><tr><td>0</td><td>2</td><td>3</td></tr><tr><td>0</td><td>3</td><td>3</td></tr></table>	σ_1	σ_2	$Freq$	0	1	3	0	2	3	0	3	3
σ_1	σ_2	$Freq$														
0	1	3														
0	2	3														
0	3	3														
\emptyset	$\{x_1, x_3\}$		$FreqTab_{\emptyset, \{x_1\}}$ $\bowtie FreqTab_{\emptyset, \{x_3\}}$	<table><tr><th>σ_1</th><th>σ_3</th><th>$Freq$</th></tr><tr><td>0</td><td>1</td><td>3</td></tr><tr><td>0</td><td>2</td><td>3</td></tr><tr><td>0</td><td>3</td><td>3</td></tr></table>	σ_1	σ_3	$Freq$	0	1	3	0	2	3	0	3	3
σ_1	σ_3	$Freq$														
0	1	3														
0	2	3														
0	3	3														
\emptyset	$\{x_2, x_3\}$		$FreqTab_{\emptyset, \{x_2\}}$ $\bowtie FreqTab_{\emptyset, \{x_3\}}$	\emptyset												
\emptyset	$\{x_1, x_2, x_3\}$		Pruned													
$\{x_1\}$	\emptyset			<table><tr><th>$Freq$</th></tr><tr><td>14</td></tr></table>	$Freq$	14										
$Freq$																
14																
$\{x_1\}$	$\{x_2\}$		$FreqTab_{\emptyset, \{x_2\}}$ $\bowtie FreqTab_{\{x_1\}, \{\emptyset\}}$	<table><tr><th>σ_2</th><th>$Freq$</th></tr><tr><td>1</td><td>3</td></tr><tr><td>2</td><td>3</td></tr><tr><td>3</td><td>3</td></tr></table>	σ_2	$Freq$	1	3	2	3	3	3				
σ_2	$Freq$															
1	3															
2	3															
3	3															
$\{x_1\}$	$\{x_3\}$		$FreqTab_{\emptyset, \{x_3\}}$ $\bowtie FreqTab_{\{x_1\}, \{\emptyset\}}$	<table><tr><th>σ_3</th><th>$Freq$</th></tr><tr><td>1</td><td>3</td></tr><tr><td>2</td><td>3</td></tr><tr><td>3</td><td>3</td></tr></table>	σ_3	$Freq$	1	3	2	3	3	3				
σ_3	$Freq$															
1	3															
2	3															
3	3															

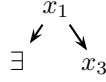
$\{x_1\}$	$\{x_2, x_3\}$	$\begin{array}{c} \exists \\ \swarrow \quad \searrow \\ \sigma_2 \quad \sigma_3 \end{array}$	$\begin{array}{l} FreqTab_{\emptyset, \{x_2, x_3\}} \\ \bowtie FreqTab_{\{x_1\}, \{x_2\}} \\ \bowtie FreqTab_{\{x_1\}, \{x_3\}} \end{array}$	\emptyset				
$\{x_2\}$	\emptyset	$\begin{array}{c} x_1 \\ \swarrow \quad \searrow \\ \exists \quad x_3 \end{array}$		<table border="1"><tr><td><i>Freq</i></td></tr><tr><td>7</td></tr></table>	<i>Freq</i>	7		
<i>Freq</i>								
7								
$\{x_2\}$	$\{x_1\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \quad \searrow \\ \exists \quad x_3 \end{array}$	$\begin{array}{l} FreqTab_{\emptyset, \{x_1\}} \\ \bowtie FreqTab_{\{x_2\}, \{\emptyset\}} \end{array}$	<table border="1"><tr><td>σ_1</td><td><i>Freq</i></td></tr><tr><td>0</td><td>3</td></tr></table>	σ_1	<i>Freq</i>	0	3
σ_1	<i>Freq</i>							
0	3							
$\{x_2\}$	$\{x_3\}$	$\begin{array}{c} x_1 \\ \swarrow \quad \searrow \\ \exists \quad \sigma_3 \end{array}$	$\begin{array}{l} FreqTab_{\emptyset, \{x_3\}} \\ \bowtie FreqTab_{\{x_2\}, \{\emptyset\}} \end{array}$	\emptyset				
$\{x_2\}$	$\{x_1, x_3\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \quad \searrow \\ \exists \quad \sigma_3 \end{array}$	Pruned					
$\{x_3\}$	\emptyset	$\begin{array}{c} x_1 \\ \swarrow \quad \searrow \\ x_2 \quad \exists \end{array}$		<table border="1"><tr><td><i>Freq</i></td></tr><tr><td>7</td></tr></table>	<i>Freq</i>	7		
<i>Freq</i>								
7								
$\{x_3\}$	$\{x_1\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \quad \searrow \\ \exists \quad x_3 \end{array}$	$\begin{array}{l} FreqTab_{\emptyset, \{x_1\}} \\ \bowtie FreqTab_{\{x_3\}, \{\emptyset\}} \end{array}$	<table border="1"><tr><td>σ_1</td><td><i>Freq</i></td></tr><tr><td>0</td><td>3</td></tr></table>	σ_1	<i>Freq</i>	0	3
σ_1	<i>Freq</i>							
0	3							
$\{x_3\}$	$\{x_2\}$	$\begin{array}{c} x_1 \\ \swarrow \quad \searrow \\ \sigma_2 \quad \exists \end{array}$	$\begin{array}{l} FreqTab_{\emptyset, \{x_2\}} \\ \bowtie FreqTab_{\{x_3\}, \{\emptyset\}} \end{array}$	\emptyset				
$\{x_3\}$	$\{x_1, x_2\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \quad \searrow \\ \sigma_2 \quad \exists \end{array}$	Pruned					
$\{x_1, x_2\}$	\emptyset	$\begin{array}{c} \exists \\ \swarrow \quad \searrow \\ \exists \quad x_3 \end{array}$		<table border="1"><tr><td><i>Freq</i></td></tr><tr><td>6</td></tr></table>	<i>Freq</i>	6		
<i>Freq</i>								
6								
$\{x_1, x_2\}$	$\{x_3\}$	$\begin{array}{c} \exists \\ \swarrow \quad \searrow \\ \exists \quad \sigma_3 \end{array}$	$\begin{array}{l} FreqTab_{\{x_1\}, \{x_3\}} \\ \bowtie FreqTab_{\{x_2\}, \{x_3\}} \\ \bowtie FreqTab_{\{x_1, x_2\}, \emptyset} \end{array}$	\emptyset				
$\{x_1, x_3\}$	\emptyset	$\begin{array}{c} \exists \\ \swarrow \quad \searrow \\ x_2 \quad \exists \end{array}$		<table border="1"><tr><td><i>Freq</i></td></tr><tr><td>6</td></tr></table>	<i>Freq</i>	6		
<i>Freq</i>								
6								

$\{x_1, x_3\}$	$\{x_2\}$	$\begin{array}{c} \exists \\ \swarrow \quad \searrow \\ \sigma_2 \quad \exists \end{array}$	$\begin{array}{l} FreqTab_{\{x_1\}, \{x_2\}} \\ \bowtie FreqTab_{\{x_3\}, \{x_2\}} \\ \bowtie FreqTab_{\{x_1, x_3\}, \emptyset} \end{array}$	\emptyset		
$\{x_2, x_3\}$	\emptyset	$\begin{array}{c} x_1 \\ \swarrow \quad \searrow \\ \exists \quad \exists \end{array}$		<table border="1"><tr><td><i>Freq</i></td></tr><tr><td>4</td></tr></table>	<i>Freq</i>	4
<i>Freq</i>						
4						
$\{x_2, x_3\}$	$\{x_1\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \quad \searrow \\ \exists \quad \exists \end{array}$	$\begin{array}{l} FreqTab_{\{x_2, x_3\}, \emptyset} \\ \bowtie FreqTab_{\{x_2\}, \{x_1\}} \\ \bowtie FreqTab_{\{x_3\}, \{x_2\}} \end{array}$	\emptyset		

4.5 Equivalence among Tree Patterns

In this Section we make a number of modifications to the algorithm described so far, so as to avoid duplicate work.

As an example of duplicate work, consider the parameterized tree pattern P_1 from the example run in Section 4.4.4 ($\Pi = \{x_2\}$ and $\Sigma = \emptyset$):

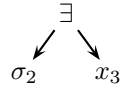


and the parameterized tree pattern P_2 :

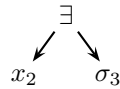


Clearly, P_1 and P_2 have the same answer set for all data graphs G , up to renaming of the distinguished variables (x_2 by x_3). However, these patterns have different underlying trees, and hence Algorithm 1 will compute the answer set for both patterns (line 6). The answer set of P_1 is computed before the answer set of P_2 , since our algorithm is incremental in the number of nodes of T . Hence, we can conclude that our algorithm performs some duplicate work which we want to avoid.

Another example of duplicate work our algorithm performs: Consider the parameterized tree pattern P_3 from the example run in Section 4.4.4 ($\Pi = \{x_1\}$ and $\Sigma = \{x_2\}$):



and the parameterized tree pattern P_4 also from the example run in Section 4.4.4 ($\Pi = \{x_1\}$ and $\Sigma = \{x_3\}$):



As one can see in Section 4.4.4, these two parameterized patterns have the same instantiations for all data graphs G , up to renaming of the parameters (σ_2 by σ_3), and for each instantiation, the same answer set for all data graphs G , up to renaming of the distinguished variables (x_3 by x_2). However, when we look at the outline of our algorithm in Algorithm 1, we see that for both patterns the candidacy and frequency tables are computed between line 11 and line 17. Hence, we can conclude again that our algorithm performs duplicate work that we want to avoid.

In the rest of this Section we formalize the duplicate work our algorithm performs, and we make a number of modifications to the algorithm described so far, so as to avoid the duplicate work.

4.5.1 Equivalency

Intuitively we call two parameterized tree patterns *equivalent* if they have the same answer sets and the same parameter assignments for all data graphs G , up to renaming of the parameters and the distinguished variables. For instance, the parameterized tree patterns P_1 and P_2 from above we call equivalent, as the tree patterns P_3 and P_4 from above.

To define equivalent parameterized tree patterns formally we introduce the notion of (δ, ρ) -equivalence.

(δ, ρ) -Equivalence Let P_1 and P_2 be two parameterized tree patterns and ρ a parameter correspondence from P_1 to P_2 (recall Section 3.3). We define an *answer set correspondence* from P_1 to P_2 as any mapping $\delta : \Delta_1 \rightarrow \Delta_2$. Furthermore, assume that δ and ρ are bijections. We then say that P_1 and P_2 are (δ, ρ) -equivalent, denoted by $P_1 \equiv_{\rho}^{\delta} P_2$, if for all data graphs G , and all parameter assignments $\alpha_2 : \Sigma_2 \rightarrow U$, we have $P_2^{\alpha_2}(G) \circ \delta = P_1^{\alpha_2 \circ \rho}(G)$, where $P_2^{\alpha_2}(G) \circ \delta$ denotes the set $\{f \circ \delta : f \in P_2^{\alpha_2}(G)\}$.

For example, consider the two parameterized tree patterns in Figure 13, and let $\rho : \Sigma_a \rightarrow \Sigma_b$ be as follows:

ρ	
σ_1	σ_1
σ_2	σ_3
σ_3	σ_2

and let $\delta : \Delta_a \rightarrow \Delta_b$ be as follows:

δ	
x_1	x_3
x_2	x_1
x_3	x_2

The two parameterized tree patterns are clearly (δ, ρ) -equivalent, as are the three parameterized tree patterns shown in Figure 14 with an empty parameter correspondence ρ and δ the identity.

The parameter correspondence ρ is a bijection in the definition of ρ -equivalence, since intuitively we want equivalent parameterized tree patterns to have essentially the same set of instantiations. Hence it is necessary that the tree patterns have the same number of parameters. Intuitively we also want equivalent

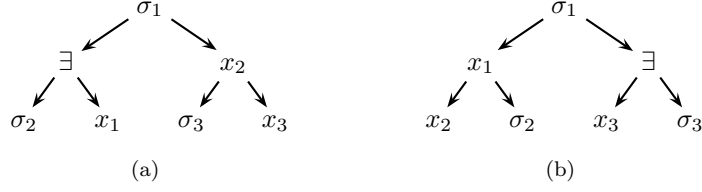


Figure 13: Two equivalent parameterized tree patterns.

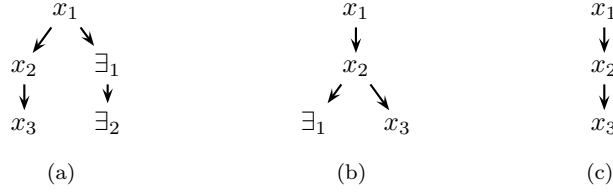


Figure 14: Three equivalent parameterized tree patterns.

tree patterns to have the same answer sets up to renaming of the distinguished variables. That is the reason why an answer set correspondence is introduced that is a bijection.

We then define equivalent parameterized tree patterns as follows:

Equivalent parameterized tree patterns We call two parameterized tree patterns P_1 and P_2 *equivalent* if P_1 is (δ, ρ) -equivalent with P_2 for some bijective parameter correspondence ρ and some bijective answer set correspondence δ .

Note that there can exist more than one parameter correspondence ρ and more than one answer set correspondence δ for which the two parameterized tree patterns are (δ, ρ) -equivalent. An illustration of this is given in Figure 15. Let $\rho_1 : \Sigma_a \rightarrow \Sigma_b$, $\delta_1 : \Delta_a \rightarrow \Delta_b$, $\rho_2 : \Sigma_a \rightarrow \Sigma_b$ and $\delta_2 : \Delta_a \rightarrow \Delta_b$ be as follows: ρ_1 is the identity; δ_1 is the identity and

<table style="border-collapse: collapse;"> <tr> <th colspan="2">ρ_2</th></tr> <tr> <td style="padding: 2px 10px;">σ_1</td><td style="padding: 2px 10px;">σ_2</td></tr> <tr> <td style="padding: 2px 10px;">σ_2</td><td style="padding: 2px 10px;">σ_1</td></tr> </table>	ρ_2		σ_1	σ_2	σ_2	σ_1	<table style="border-collapse: collapse;"> <tr> <th colspan="2">δ_2</th></tr> <tr> <td style="padding: 2px 10px;">x_1</td><td style="padding: 2px 10px;">x_1</td></tr> <tr> <td style="padding: 2px 10px;">x_2</td><td style="padding: 2px 10px;">x_4</td></tr> <tr> <td style="padding: 2px 10px;">x_3</td><td style="padding: 2px 10px;">x_5</td></tr> <tr> <td style="padding: 2px 10px;">x_4</td><td style="padding: 2px 10px;">x_2</td></tr> <tr> <td style="padding: 2px 10px;">x_5</td><td style="padding: 2px 10px;">x_3</td></tr> </table>	δ_2		x_1	x_1	x_2	x_4	x_3	x_5	x_4	x_2	x_5	x_3
ρ_2																			
σ_1	σ_2																		
σ_2	σ_1																		
δ_2																			
x_1	x_1																		
x_2	x_4																		
x_3	x_5																		
x_4	x_2																		
x_5	x_3																		

Then the two tree patterns in Figure 15 are clearly (δ_1, ρ_1) -equivalent and (δ_2, ρ_2) -equivalent.

Equivalence as just defined is a semantical property, referring to all possible data graphs, and it is not immediately clear how one could decide this property syntactically. The required syntactical notion is given by the following Lemma and Corollary.

Lemma 4. *Consider two parameterized tree patterns P_1 and P_2 , $\delta : \Delta_1 \rightarrow \Delta_2$ a bijective answer set correspondence, and $\rho : \Sigma_1 \rightarrow \Sigma_2$ a bijective parameter*



Figure 15: Two equivalent parameterized tree patterns with more than one possibility for a parameter and answer set correspondence.

correspondence. Then $P_1 \equiv_\rho^\delta P_2$ if and only if we have the following containment relations among the tree queries (H_1, P_1) and $(\delta(H_1), P_2)$, with H_1 the pure head of P_1 (cfr. Section 4.1):

1. $(\delta(H_1), P_2) \subseteq_\rho (H_1, P_1)$; and
2. $(H_1, P_1) \subseteq_{\rho^{-1}} (\delta(H_1), P_2)$

Proof. Let us start with the if direction. We need to prove that for every parameter assignment α_2 for P_2 , and every data graph G that $P_2^{\alpha_2}(G) \circ \delta = P_1^{\alpha_2 \circ \rho}(G)$. We know that $(\delta(H_1), P_2)^{\alpha_2}(G) \subseteq (H_1, P_1)^{\alpha_2 \circ \rho}(G)$ since $(\delta(H_1), P_2) \subseteq_\rho (H_1, P_1)$. We may rewrite this as: $P_2^{\alpha_2}(G) \circ \delta \subseteq P_1^{\alpha_2 \circ \rho}(G)$ since H_1 is an enumeration of Δ_1 .

We also know that $(H_1, P_1)^{\alpha_1}(G) \subseteq (\delta(H_1), P_2)^{\alpha_1 \circ \rho^{-1}}(G)$ for every parameter assignment α_1 for P_1 since $(H_1, P_1) \subseteq_{\rho^{-1}} (\delta(H_1), P_2)$. Now take $\alpha_1 = \alpha_2 \circ \rho$. We then have $(H_1, P_1)^{\alpha_2 \circ \rho}(G) \subseteq (\delta(H_1), P_2)^{\alpha_2}(G)$. Again since H_1 is an enumeration of Δ_1 we may rewrite this as: $P_1^{\alpha_2 \circ \rho}(G) \subseteq P_2^{\alpha_2}(G) \circ \delta$. Hence we can conclude that $P_2^{\alpha_2}(G) \circ \delta = P_1^{\alpha_2 \circ \rho}(G)$.

Let us then look at the only-if direction. To prove that $(\delta(H_1), P_2) \subseteq_\rho (H_1, P_1)$, we will show that for every α_2 parameter assignment for P_2 , and every data graph G , we have $(\delta(H_1), P_2)^{\alpha_2}(G) \subseteq (H_1, P_1)^{\alpha_2 \circ \rho}(G)$. Since $P_2^{\alpha_2}(G) \circ \delta = P_1^{\alpha_2 \circ \rho}(G)$, we have $(\delta(H_1), P_2)^{\alpha_2}(G) = (H_1, P_1)^{\alpha_2 \circ \rho}(G)$, and hence clearly $(\delta(H_1), P_2)^{\alpha_2}(G) \subseteq (H_1, P_1)^{\alpha_2 \circ \rho}(G)$.

To prove that $(H_1, P_1) \subseteq_{\rho^{-1}} (\delta(H_1), P_2)$, we will show that for every α_1 parameter assignment for P_1 , and every data graph G , we have $(H_1, P_1)^{\alpha_1}(G) \subseteq (\delta(H_1), P_2)^{\alpha_1 \circ \rho^{-1}}(G)$. We know that for every α_2 parameter assignment for P_2 , we have $P_2^{\alpha_2}(G) \circ \delta = P_1^{\alpha_2 \circ \rho}(G)$. Now take $\alpha_2 = \alpha_1 \circ \rho^{-1}$. We then have $P_2^{\alpha_1 \circ \rho^{-1}}(G) \circ \delta = P_1^{\alpha_1}(G)$, hence $(\delta(H_1), P_2)^{\alpha_1 \circ \rho^{-1}}(G) = (H_1, P_1)^{\alpha_1}(G)$, hence clearly $(H_1, P_1)^{\alpha_1}(G) \subseteq (\delta(H_1), P_2)^{\alpha_1 \circ \rho^{-1}}(G)$. \square

Corollary 1. Consider two parameterized tree patterns P_1 and P_2 . Then P_1 is equivalent with P_2 if and only if there exist:

1. a bijective answer set correspondence $\delta : \Delta_1 \rightarrow \Delta_2$;
2. a bijective parameter correspondence $\rho : \Sigma_1 \rightarrow \Sigma_2$;
3. a ρ -containment mapping $f_1 : (H_1, P_1) \rightarrow (\delta(H_1), P_2)$; and
4. a ρ^{-1} -containment mapping $f_2 : (\delta(H_1), P_2) \rightarrow (H_1, P_1)$.

with H_1 the pure head for P_1 .

Of course, we want to avoid that our algorithm considers some parameterized tree pattern P_2 if it is equivalent to an earlier considered parameterized tree pattern P_1 . Since our algorithm generates trees in increasing sizes, there are two cases to consider:

Case A: P_1 has fewer nodes than P_2 .

Case B: P_1 and P_2 have the same number of nodes.

Armed with the above Lemma and Corollary, we can now analyze the above two cases.

4.5.2 Case A: Redundancy checking

Let us start by defining the notion of a redundancy.

Redundant subtree A *redundant subtree* R , is a subtree of a parameterized tree pattern P , such that removing R from P yields a parameterized tree pattern P' that is equivalent with P .

For example, the first two parameterized tree patterns in Figure 14 indeed contain a redundant subtree.

The following lemma shows that two parameterized tree patterns with different numbers of nodes can only be equivalent if the largest one contains redundant subtrees:

Lemma 5. *Consider two parameterized tree patterns P and P' , and the number of nodes of P' is smaller than the number of nodes of P . Then P can only be equivalent with P' if P contains redundant subtrees.*

Proof. Since P and P' are equivalent we know from Corollary 1 that the following exist:

1. an answer set correspondence $\delta : \Delta \rightarrow \Delta'$ that is a bijection;
2. a parameter correspondence $\rho : \Sigma \rightarrow \Sigma'$ that is a bijection;
3. a ρ -containment mapping $f_1 : (H, P) \rightarrow (\delta(H), P')$; and
4. a ρ^{-1} -containment mapping $f_2 : (\delta(H), P') \rightarrow (H, P)$.

with H the pure head for P . Since the number of nodes of P' is smaller than the number of nodes of P , we know that some subtrees R of P are not in the range of f_2 . We will prove that these subtrees R are redundant subtrees, by showing that P and $P - R$ are equivalent.

Since the containment mappings f_1 and f_2 exist, we know that in particular the following containment mappings will exist:

1. $g_1 = f_1|_{P-R}$, a ρ -containment mapping from $(H, P - R)$ to $(\delta(H), P')$, and
2. $g_2 = f_2$, a ρ^{-1} -containment mapping from $(\delta(H), P')$ to $(H, P - R)$.

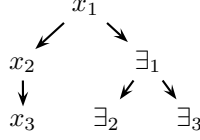


Figure 16: A tree pattern that contains a linear chain of existential nodes that is redundant.

with δ and ρ as above.

Let us now look at the following mappings: $h_1 = g_2 \circ f_1$ and $h_2 = f_2 \circ g_1$. By Lemma 1, $h_1 = g_2 \circ f_1$ and $h_2 = f_2 \circ g_1$ are identity-containment mappings.

Using Corollary 1 we can now conclude that P and $P - R$ are (identity, identity)-equivalent and thus R is a redundant subtree. \square

From this lemma follows that Case A can only happen if P_2 contains redundant subtrees. Hence, if we can avoid redundancies, Case A will never occur.

The following lemma provides us with an efficient check for redundancies.

Redundancy Lemma. *Let P be a parameterized tree pattern. Then P has a redundancy if and only if it contains a subtree C in the form of a linear chain of existential nodes (possibly just a single node), such that the parent of C has another subtree that is at least as deep as C .*

Before we prove this Lemma, let us see some examples. For instance the parameterized tree patterns in Figure 14(a) and Figure 14(b) contain a linear chain of existential nodes that is redundant. In both tree patterns this linear chain is rooted in \exists_1 . Another example of such a redundancy is given in Figure 16. Here the linear chain is rooted in \exists_3 . Note that when we remove the linear chain rooted in \exists_3 , we have a new linear chain rooted in \exists_1 that is redundant.

Proof. Let us refer to a subtree C as described in the lemma as an “eliminable path”. An eliminable path is clearly redundant, so we only need to prove the only-if direction. Let T be a redundant subtree of P that is maximal, in the sense that it is not the subtree of another redundant subtree. Then following Corollary 1, there must be a ρ -containment mapping h from (H, P) to $(\delta(H), P - T)$ with ρ and δ bijections and H the pure head for P . All distinguished variables of P must be in $P - T$, since δ is a bijection. Also all parameters of P must be in $P - T$, since ρ is also a bijection. So T consists entirely of existential nodes.

Furthermore, note that h must fix the root of P , since the height of P is at least that of $P - T$.

Any iteration h^n of h is a ρ^n -containment mapping from (H, P) to $(\delta^n(H), P - T)$ by Lemma 1. Moreover, each $h^n|_{\Delta \cup \Sigma}$ induces a permutation on the set $\Delta \cup \Sigma$ of distinguished variables and parameters. Since $\Delta \cup \Sigma$ is finite, there are only a finite number of possible permutations of $\Delta \cup \Sigma$, namely $|\Delta \cup \Sigma|!$. Hence, there will be an iteration $h^k|_{\Delta \cup \Sigma}$ and an iteration $h^{(k+l)}|_{\Delta \cup \Sigma}$ such that

$h^k|_{\Delta \cup \Sigma} = h^{(k+l)}|_{\Delta \cup \Sigma}$. Hence, $h^l|_{\Delta \cup \Sigma}$ is the identity on $\Delta \cup \Sigma$, because

$$\begin{aligned} \text{id}|_{\Delta \cup \Sigma} &= (h^{-1})^k|_{\Delta \cup \Sigma} \circ h^k|_{\Delta \cup \Sigma} \\ &= (h^{-1})^k|_{\Delta \cup \Sigma} \circ h^{(k+l)}|_{\Delta \cup \Sigma} \\ &= h^l|_{\Delta \cup \Sigma} \end{aligned}$$

There are now two possible cases.

1. T itself is a linear chain. Let us then look at the parent p of T in P . Again there are two possibilities:

(a) $h^l(p) = p$: Since h^l is a ρ^l -containment mapping from (H, P) to $(\delta^l(H), P - T)$ and T is redundant, we know that T must be mapped to another subtree of p , T' , that is at least as deep as T . Hence, T is an eliminable path. An illustration is given in Figure 17(a).

(b) $h^l(p) \neq p$: Then p can only be an existential node. We now have two possibilities:

i. T is the only subtree of p . We will show that the subtree T' , rooted in p is redundant as well. Clearly we have the following containment relations:

- $h_1 = h^l$, a ρ^l -containment mapping from (H, P) to $(\delta^l(H), P - T')$; and
- $h_2 = h^{-l}$, a ρ^{-l} -containment mapping from $(\delta^l(H), P - T')$ to (H, P) .

with δ and ρ as above. By Corollary 1, T' is then a redundant subtree. This is in contraction with the assumption that T is maximal. Hence, it is impossible that p has only one subtree and p is existential. An illustration of this is in given in Figure 17(b).

ii. p has more than one subtree. Consider such another subtree T' . We will show that all subtrees T' of p consist entirely of existential nodes. Suppose a node $n \in T'$ is not an existential node. We then know that $h^l(n) = n$. However, since h^l is a homomorphism and p is an ancestor of n , $h^l(p)$ must be p . But this is in contradiction with the assumption that $h^l(p) \neq p$. So T' must consist entirely of existential nodes. Hence this brings us to the second case where T is not a linear chain. An illustration is given in Figure 17(c).

2. T is not a linear chain. An easy induction on the height of T , shows that any non-linear tree consisting entirely of existential nodes must contain an eliminable path. If the height of T is 1, there is an eliminable path of a single node: just choose one of the children of the root. If the height of T is $n > 1$, consider the subtree S of the root of T with the smallest height, at most $n - 1$. Then we have two possibilities: If S is a linear chain, we found our eliminable path. And if S is a non-linear chain we know by induction that S will contain an eliminable path. Hence, T , and thus also P , contains an eliminable path as desired.

□

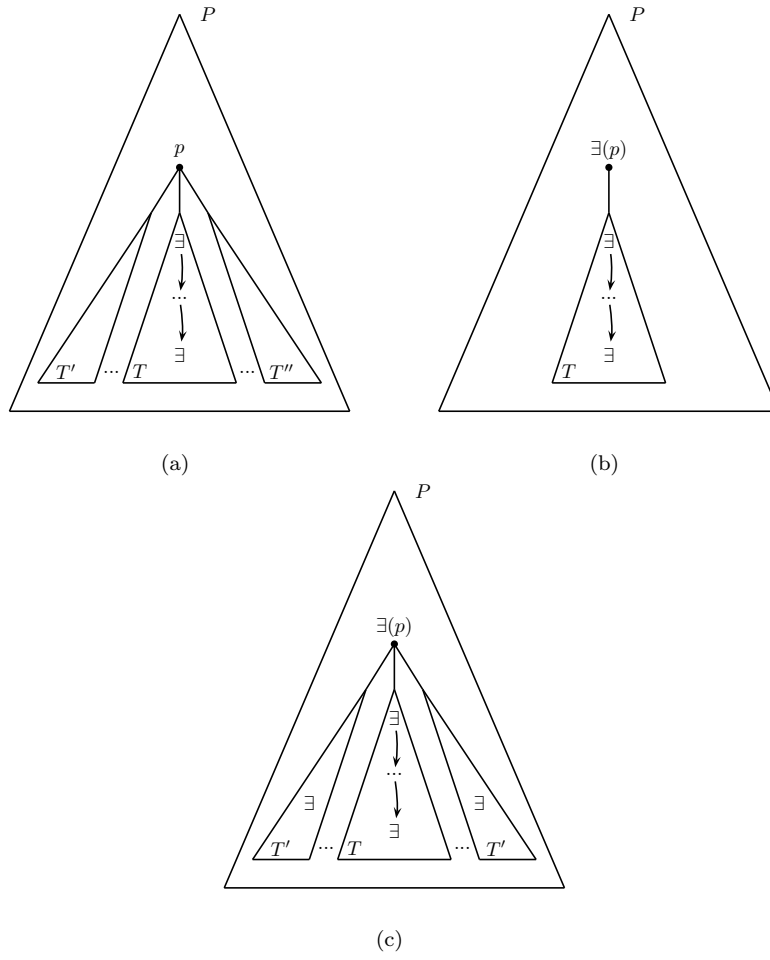


Figure 17: Figures to illustrate the proof of the Redundancy Lemma.

As we have seen in Section 4.4, our algorithm introduces existential nodes levelwise, one by one. This makes the redundancy test provided by the redundancy lemma particularly easy to perform. Indeed, if (Π, Σ) is a parameterized tree patterns of which we already know it has no redundancies, and we make one additional node n existential, then it suffices to test whether n thus becomes part of a subtree C as in the Redundancy Lemma. If so, we will prune the entire search at $\Pi \cup \{n\}$.

4.5.3 Case B: Canonical forms

We may now assume that P_1 and P_2 do not contain redundancies, for if they would, they would have been dismissed already.

Let us start by defining isomorphic parameterized tree patterns.

Isomorphic Parametrized Tree Patterns We call two parameterized tree patterns P_1 and P_2 *isomorphic* if there exists a homomorphism $f : P_1 \rightarrow P_2$ that is a bijection and that maps distinguished nodes to distinguished nodes, parameters to parameters and existential nodes to existential nodes. We call f an *isomorphism*. Since we are working with trees, f^{-1} is also a homomorphism.

For example, the two parameterized tree patterns in Figure 13 are indeed isomorphic with f as follows:

f	
σ_1	σ_1
\exists	\exists
σ_2	σ_3
x_1	x_3
x_2	x_1
σ_3	σ_2
x_3	x_2

Clearly, we have the following:

Property 1. *Any two isomorphic parameterized tree patterns P_1 and P_2 are equivalent.*

Proof. Using Corollary 1 we have to show that the following exists:

1. a bijective answer set correspondence $\delta : \Delta_1 \rightarrow \Delta_2$;
2. a bijective parameter correspondence $\rho : \Sigma_1 \rightarrow \Sigma_2$;
3. a ρ -containment mapping $f_1 : (H_1, P_1) \rightarrow (\delta(H_1), P_2)$; and
4. a ρ^{-1} -containment mapping $f_2 : (\delta(H_1), P_2) \rightarrow (H_1, P_1)$.

with H_1 the pure head for P_1 .

Since P_1 and P_2 are isomorphic, there exists a homomorphism $f : P_1 \rightarrow P_2$ that is a bijection and that maps distinguished nodes to distinguished nodes, parameters to parameters and existential nodes to existential nodes.

We now take $\delta = f|_{\Delta_1}$ and $\rho = f|_{\Sigma_1}$. Then δ and ρ are bijections since f is a bijection.

For (3) we will show that f is ρ -containment mapping from (H_1, P_1) to $(\delta(H_1), P_2)$, with H_1 the pure head for P_1 :

- f is a homomorphism;
- f maps distinguished nodes to distinguished nodes and $f|_{\Delta_1} = \delta$;
- f maps parameters to parameters and $f|_{\Sigma_1} = \rho$; and
- $f(H_1) = \delta(H_1)$.

For (4) we will show that f^{-1} is ρ^{-1} -containment mapping from $(\delta(H_1), P_2)$ to (H_1, P_1) , with H_1 the pure head for P_1 :

- f^{-1} is a homomorphism since f is a bijection and we are working with trees;
- f^{-1} maps distinguished nodes to distinguished nodes and $f^{-1}|_{\Delta_2} = \delta^{-1}$;
- f^{-1} maps parameters to parameters and $f^{-1}|_{\Sigma_2} = \rho^{-1}$; and
- $f^{-1}(\delta(H_1)) = H_1$.

□

The following lemma shows that two parameterized tree patterns without redundancies and with the same number of nodes can only be equivalent if they are isomorphic.

Isomorphism Lemma. *Consider two parameterized tree patterns P_1 and P_2 without redundancies, and with the same number of nodes. Then P_1 and P_2 are equivalent if and only if P_1 and P_2 are isomorphic.*

Proof. We only need to show the only-if direction.

Since P_1 and P_2 are equivalent we know that the following exists by Corollary 1:

1. a bijective answer set correspondence $\delta : \Delta_1 \rightarrow \Delta_2$;
2. a bijective parameter correspondence $\rho : \Sigma_1 \rightarrow \Sigma_2$;
3. a ρ -containment mapping $f_1 : (H_1, P_1) \rightarrow (\delta(H_1), P_2)$; and
4. a ρ^{-1} -containment mapping $f_2 : (\delta(H_1), P_2) \rightarrow (H_1, P_1)$.

with H_1 a pure head for P_1 .

We also know that P_1 and P_2 have the same number of existential nodes since P_1 and P_2 have the same number of nodes and ρ and δ are bijections.

Hence to prove that P_1 and P_2 are isomorphic, we only need to show that:

1. f_1 maps existential nodes to existential nodes; and
2. $f_1|_{\Pi_1}$ is a bijection.

Thereto, it suffices to prove that f_1 is surjective on the existential nodes of P_2 , because f_1 is already a bijection from $\Delta_1 \cup \Sigma_1$ to $\Delta_2 \cup \Sigma_2$.

Assume that $f_1|_{\Pi_1}$ is not surjective. Hence, there will be some existential nodes $p \in \Pi_2$ that are not in the range of f_1 . Note that these existential nodes p can never have descendants that are parameters or distinguished nodes since f_1 is a homomorphism and δ and ρ bijections. Now fix some p as high as possible

in the tree. Then the entire subtree R rooted in p consists entirely of existential nodes, that are not in the range of f_1 . We will now show that this subtree R is a redundant subtree in P_2 . We then have a contradiction since we assume that P_2 is redundancy free.

Since the containment mappings f_1 and f_2 exist, we know that in particular the following containment mappings will exist:

1. $g_1 = f_1$ a ρ -containment mapping from (H_1, P_1) to $(\delta(H_1), P_2 - R)$, and
2. $g_2 = f_2|_{P_2 - R}$ a ρ^{-1} -containment mapping from $(\delta(H_1), P_2 - R)$ to (H_1, P_1) .

with δ and ρ as above.

Let us now look at the following mappings $h_1 = f_1 \circ g_2$ and $h_2 = g_1 \circ f_2$. By Lemma 1, h_1 and h_2 are identity-containment mappings. Using Corollary 1, we can now conclude that P_2 and $P_2 - R$ are equivalent, and thus R is a redundant subtree. \square

From the above lemma it follows that Case B can only happen if P_1 and P_2 are actually isomorphic. In particular, P_1 and P_2 have the same underlying tree.

So, in our algorithm, we need an efficient way to avoid isomorphic parameterized tree patterns based on the same tree T .

Fortunately, there is a standard way to do this, by working with *canonical forms* of parameterized tree patterns. Consider a pair (Π, Σ) , as in Section 4.4. We can view this pair as a labeling of T : all nodes in Π get the same generic label ' \exists '; all nodes in Σ get ' σ '; and all distinguished nodes get ' x '. We then observe that the patterns (Π_1, Σ_1) and (Π_2, Σ_2) are isomorphic iff there is a tree isomorphism between the corresponding labeled versions of T that respects the labels.

In order to represent each pair (Π, Σ) uniquely up to isomorphism, we can rather straightforwardly refine the canonical ordering of the underlying unlabeled tree T , which we already have (Section 4.3), to take into account the node labels. Furthermore, the classical linear-time algorithm to canonize a tree [4] generalizes straightforwardly to labeled trees. A nice review of these generalizations has been given by Chi, Yang and Muntz [10].

We will omit the details of the canonical form; in fact, there are several ways to realize it. All that is important is that we can check in linear time whether a pair is canonical; that a pair can be canonized in linear time; and that two pairs are isomorphic if and only if their canonical forms are identical.

Example. We can refine the level sequence introduced in Section 4.3 to a *refined level sequence* for parameterized tree patterns P as follows: if the tree pattern P consists of n nodes, then the *refined level sequence* is now a sequence of n elements, where the i th element is the depth of the i th node in preorder in the pattern, followed by a 'd' if the node is distinguished; followed by a 'e' if the node is existential and followed by a 'p' if the node is a parameter. The canonical ordering of a parameterized tree pattern P , is then the ordering of P that yields the lexicographically maximal refined level sequence, among all orderings of P . Then the refined level sequences for the parameterized tree patterns in Figure 13 are:

- (a) 0p1e2p2d1d2p2d

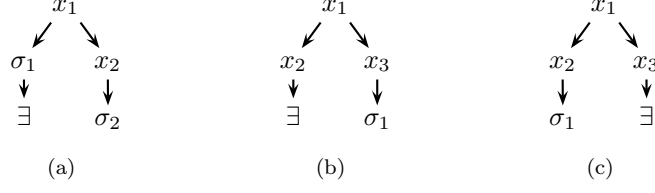


Figure 18: The tree pattern in (b) is a parent of a tree pattern in (a), and (c) is the canonical version of (b).

(b) 0p1d2d2p1e2d2p

and (a) is the canonical one.

Armed by the canonical form, we are now in a position to describe how the algorithm of Section 4.4 must be modified to avoid equivalent parameterized tree patterns. First of all, we only work with patterns (Π, Σ) in canonical form; the others are dismissed. However, the problem then arises that a parent pattern (Π', Σ') , where we omit a variable from either Π or Σ as described in Section 4.4, might be non-canonical. In that case the frequency table for (Π', Σ') will not exist. We can solve this by canonizing (Π', Σ') to its canonical version (Π'', Σ'') , and remembering the renaming of variables this entails. The table $FreqTab_{\Pi'', \Sigma''}$ can then serve in place of $FreqTab_{\Pi', \Sigma'}$, after we have applied the inverse renaming to its column headings.

This does not completely solve the problem, however. Indeed the frequency table of (Π'', Σ'') might not yet have been computed. For example, consider the parameterized tree pattern in Figure 18(a), and one of its parents in Figure 18(b). The canonical version of this parent, using the canonical ordering from the previous example, is shown in Figure 18(c). Using the current order for computing the frequency tables as in Algorithm 1, the frequency table for the pattern in Figure 18(c) is not yet computed, when we want to compute the frequency for the pattern in Figure 18(a).

We can solve this by changing the order in which we compute the frequency tables. We work with increasing levels: in level i we compute the frequency tables for all pairs (Π, Σ) , where $\#\Pi + \#\Sigma = i$. If we use this order, we are sure that when we compute the frequency table of a pair (Π, Σ) , all frequency tables of pairs (Π', Σ') with $(\#\Pi' + \#\Sigma') < (\#\Pi + \#\Sigma)$, have been computed.

4.5.4 The Algorithm

The final algorithm is now given in Algorithm 2. The outline for canonizing a parameterized tree pattern is given in Function 3.

4.5.5 Example run

In this Section we give an example run of the final algorithm in Algorithm 2. We use the same data graph G ; unordered rooted tree T ; and minimum support threshold, 3, as in the example in Section 4.4.4.

Algorithm 2 Levelwise search for frequent tree patterns with equivalence checking.

```

1: for each unordered, rooted tree  $T$  do
2:   level := number of nodes of  $T$ 
3:    $i := 0$ 
4:    $C_0 := \{(\emptyset, \emptyset)\}; F := \emptyset$ 
5:   while  $i \leq \text{level}$  AND  $C_i \neq \emptyset$  do
6:     {Candidate evaluation}
7:     for each pattern  $(\Pi, \Sigma)$  in  $C_i$  do
8:       if  $\Sigma = \emptyset$  then
9:         Compute  $\text{FreqTab}_{\Pi, \emptyset}$  in SQL
10:      else
11:        if  $(\#\Sigma = 1 \text{ AND } \#\Pi = 0)$  then
12:           $\text{CanTab}_{\Pi, \Sigma} := \text{set of nodes of } G$ 
13:        else
14:           $\text{CanTab}_{\Pi, \Sigma} := \bowtie \{f^{-1}(\text{FreqTab}_{\Pi'', \Sigma''}) \mid (\Pi', \Sigma') \text{ parent of } (\Pi, \Sigma)$ 
15:                                     and  $(f, (\Pi'', \Sigma'')) = \text{Canonize}(\Pi', \Sigma')\}$ 
16:        end if
17:      end if
18:      Compute  $\text{FreqTab}_{\Pi, \Sigma}$  in SQL
19:      if  $(\text{FreqTab}_{\Pi, \Sigma} \neq \emptyset)$  then
20:         $F = F \cup \{(\Pi, \Sigma)\}$ 
21:      end if
22:    end for
23:    {Candidate generation}
24:     $C_{i+1} = \{(\Pi, \Sigma) \mid \text{all parents } (\Pi', \Sigma') \text{ of } (\Pi, \Sigma) \text{ are in } F\}$ 
25:    {Equivalence Check}
26:    for each pattern  $(\Pi, \Sigma)$  in  $C_{i+1}$  do
27:      if  $((\Pi, \Sigma) \text{ contains a redundancy})$  then
28:        remove  $(\Pi, \Sigma)$  from  $C_{i+1}$ 
29:      else if  $((\Pi, \Sigma) \text{ is not canonical})$  then
30:        remove  $(\Pi, \Sigma)$  from  $C_{i+1}$ 
31:      end if
32:    end for
33:     $i := i + 1$ 
34:  end while
35: end for

```

Function 3 Canonize (Π', Σ') based on T

```

1:  $(\Pi_c, \Sigma_c) := \text{canonization of } (\Pi', \Sigma') \text{ based on } T$ 
2:  $f := \text{isomorphism from } (\Pi_c, \Sigma_c) \text{ to } (\Pi', \Sigma')$ 
3: return  $(f, (\Pi_c, \Sigma_c))$ 

```




Note that there are two important differences between this run and the run in Section 4.4.4:

1. duplicate work is avoided: equivalent tree patterns are not generated; and
2. the order for computing the tree patterns is different in the sense that here, the tree patterns are generated in levels, as explained in Section 4.5.3.

The example run then looks as follows:

Π	Σ	P	$CanTab$	$FreqTab$												
Level 0																
\emptyset	\emptyset	$\begin{array}{c} x_1 \\ \swarrow \quad \searrow \\ x_2 \quad x_3 \end{array}$		<table><tr><td>$Freq$</td></tr><tr><td>15</td></tr></table>	$Freq$	15										
$Freq$																
15																
Level 1																
\emptyset	$\{x_1\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \quad \searrow \\ x_2 \quad x_3 \end{array}$	nodes of G	<table><tr><td>σ_1</td><td>$Freq$</td></tr><tr><td>0</td><td>9</td></tr><tr><td>2</td><td>4</td></tr></table>	σ_1	$Freq$	0	9	2	4						
σ_1	$Freq$															
0	9															
2	4															
\emptyset	$\{x_2\}$	$\begin{array}{c} x_1 \\ \swarrow \quad \searrow \\ \sigma_2 \quad x_3 \end{array}$	nodes of G	<table><tr><td>σ_2</td><td>$Freq$</td></tr><tr><td>1</td><td>3</td></tr><tr><td>2</td><td>3</td></tr><tr><td>3</td><td>3</td></tr><tr><td>4</td><td>3</td></tr></table>	σ_2	$Freq$	1	3	2	3	3	3	4	3		
σ_2	$Freq$															
1	3															
2	3															
3	3															
4	3															
\emptyset	$\{x_3\}$	$\begin{array}{c} x_1 \\ \swarrow \quad \searrow \\ x_2 \quad \sigma_3 \end{array}$	Equivalent with $(\emptyset, \{x_2\})$													
$\{x_1\}$	\emptyset	$\begin{array}{c} \exists \\ \swarrow \quad \searrow \\ x_2 \quad x_3 \end{array}$		<table><tr><td>$Freq$</td></tr><tr><td>14</td></tr></table>	$Freq$	14										
$Freq$																
14																
$\{x_2\}$	\emptyset	$\begin{array}{c} x_1 \\ \swarrow \quad \searrow \\ \exists \quad x_3 \end{array}$	Redundancy													
$\{x_3\}$	\emptyset	$\begin{array}{c} x_1 \\ \swarrow \quad \searrow \\ x_2 \quad \exists \end{array}$	Equivalent with $(\{x_2\}, \emptyset)$													
Level 2																
\emptyset	$\{x_1, x_2\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \quad \searrow \\ \sigma_2 \quad x_3 \end{array}$	$FreqTab_{\emptyset, \{x_1\}}$ $\bowtie FreqTab_{\emptyset, \{x_2\}}$	<table><tr><td>σ_1</td><td>σ_2</td><td>$Freq$</td></tr><tr><td>0</td><td>1</td><td>3</td></tr><tr><td>0</td><td>2</td><td>3</td></tr><tr><td>0</td><td>3</td><td>3</td></tr></table>	σ_1	σ_2	$Freq$	0	1	3	0	2	3	0	3	3
σ_1	σ_2	$Freq$														
0	1	3														
0	2	3														
0	3	3														
\emptyset	$\{x_1, x_3\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \quad \searrow \\ x_2 \quad \sigma_3 \end{array}$	Equivalent with $(\emptyset, \{x_1, x_2\})$													
\emptyset	$\{x_2, x_3\}$	$\begin{array}{c} x_1 \\ \swarrow \quad \searrow \\ \sigma_2 \quad \sigma_3 \end{array}$	$FreqTab_{\emptyset, \{x_2\}}$ $\bowtie FreqTab_{\emptyset, \{x_3\}}$	\emptyset												

$\{x_1\}$	$\{x_2\}$	$\begin{array}{c} \exists \\ \swarrow \quad \searrow \\ \sigma_2 \quad x_3 \end{array}$	$FreqTab_{\emptyset, \{x_2\}}$ $\bowtie FreqTab_{\{x_1\}, \{\emptyset\}}$	<table><tr><th>σ_2</th><th>$Freq$</th></tr><tr><td>1</td><td>3</td></tr><tr><td>2</td><td>3</td></tr><tr><td>3</td><td>3</td></tr></table>	σ_2	$Freq$	1	3	2	3	3	3
σ_2	$Freq$											
1	3											
2	3											
3	3											
$\{x_1\}$	$\{x_3\}$	$\begin{array}{c} \exists \\ \swarrow \quad \searrow \\ x_2 \quad \sigma_3 \end{array}$	Equivalent with $(\{x_1\}, \{x_2\})$									
$\{x_2\}$	$\{x_1\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \quad \searrow \\ \exists \quad x_3 \end{array}$	Redundancy									
$\{x_2\}$	$\{x_3\}$	$\begin{array}{c} x_1 \\ \swarrow \quad \searrow \\ \exists \quad \sigma_3 \end{array}$	Redundancy									
$\{x_3\}$	$\{x_1\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \quad \searrow \\ x_2 \quad \exists \end{array}$	Equivalent with $(\{x_2\}, \{x_1\})$									
$\{x_3\}$	$\{x_2\}$	$\begin{array}{c} x_1 \\ \swarrow \quad \searrow \\ \sigma_2 \quad \exists \end{array}$	Equivalent with $(\{x_2\}, \{x_3\})$									
$\{x_1, x_2\}$	\emptyset	$\begin{array}{c} \exists \\ \swarrow \quad \searrow \\ \exists \quad x_3 \end{array}$	Redundancy									
$\{x_1, x_3\}$	\emptyset	$\begin{array}{c} \exists \\ \swarrow \quad \searrow \\ x_2 \quad \exists \end{array}$	Equivalent with $(\{x_1, x_2\}, \emptyset)$									
$\{x_2, x_3\}$	\emptyset	$\begin{array}{c} x_1 \\ \swarrow \quad \searrow \\ \exists \quad \exists \end{array}$	Redundancy									
Level 3												
\emptyset	$\{x_1, x_2, x_3\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \quad \searrow \\ \sigma_2 \quad \sigma_3 \end{array}$	Pruned									
$\{x_1\}$	$\{x_2, x_3\}$	$\begin{array}{c} \exists \\ \swarrow \quad \searrow \\ \sigma_2 \quad \sigma_3 \end{array}$	$FreqTab_{\emptyset, \{x_2, x_3\}}$ $\bowtie FreqTab_{\{x_1\}, \{x_2\}}$ $\bowtie FreqTab_{\{x_1\}, \{x_3\}}$	\emptyset								
$\{x_2\}$	$\{x_1, x_3\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \quad \searrow \\ \exists \quad \sigma_3 \end{array}$	Pruned									
$\{x_3\}$	$\{x_1, x_2\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \quad \searrow \\ \sigma_2 \quad \exists \end{array}$	Equivalent with $(\{x_2\}, \{x_1, x_3\})$									

$\{x_1, x_2\}$	$\{x_3\}$		Redundancy
$\{x_1, x_3\}$	$\{x_2\}$		Equivalent with $(\{x_1, x_2\}, \{x_3\})$
$\{x_2, x_3\}$	$\{x_1\}$		Redundancy

4.6 Result Management: Pattern Database

When the algorithm is terminated, its final output consists of a set of frequency tables for each tree T that was investigated. All frequency tables are kept in a relational database that we call the *pattern database*. The pattern database is an ideal platform for an interactive tool for browsing the frequent queries. We developed such a browser called *Certhia* and discuss it in Section 6.

The pattern database is also an ideal platform for tree query association mining as will be described in Section 5.

5 Mining Tree-Query Associations

In this Section we present an algorithm for mining confident tree-query associations in a large data graph. Recall from Section 3.4 that a parameterized association rule (pAR) is something of the form $Q_1 \Rightarrow_\rho Q_2$, with Q_1 and Q_2 parameterized tree queries, $\rho : \Sigma_1 \rightarrow \Sigma_2$ a parameter correspondence, and $Q_2 \subseteq_\rho Q_1$. An instantiated association rule (iAR) is a pair $(Q_1 \Rightarrow_\rho Q_2, \alpha)$, with $Q_1 \Rightarrow_\rho Q_2$ a pAR and $\alpha : \Sigma_2 \rightarrow U$ a parameter assignment for $Q_1 \Rightarrow_\rho Q_2$. Also recall that the confidence of an iAR in a data graph G is defined as $\text{Freq}(Q_2^{\alpha_2}) / \text{Freq}(Q_1^{\alpha_2 \circ \rho})$.

The algorithm presented in this Section finds all iARs of the form $(Q_{\text{left}} \Rightarrow_\rho Q_{\text{right}}, \alpha)$ that are confident and frequent in a given data graph G for a given lhs Q_{left} . Before presenting the algorithm we first show that we do not need to tackle the problem in its full generality.

5.1 Problem Reduction

In this section we show that, without loss of generality, we can focus on the case where the given lhs tree query Q_{left} is pure in the sense that was defined in Section 4.1. We will also show that this restriction can not be imposed on the rhs tree queries to be output. We also make a remark regarding “free constants” in the head of a tree query.

Pure lhs’s Assume that all possible variables (nodes of tree patterns) have been arranged in some fixed but arbitrary order. Recall then from Section 4.1 that we call a parameterized tree query $Q = (H, P)$ pure when H consists of the enumeration in order and without repetitions of all distinguished variables of P . In particular H can not contain parameters. We call H the pure head for

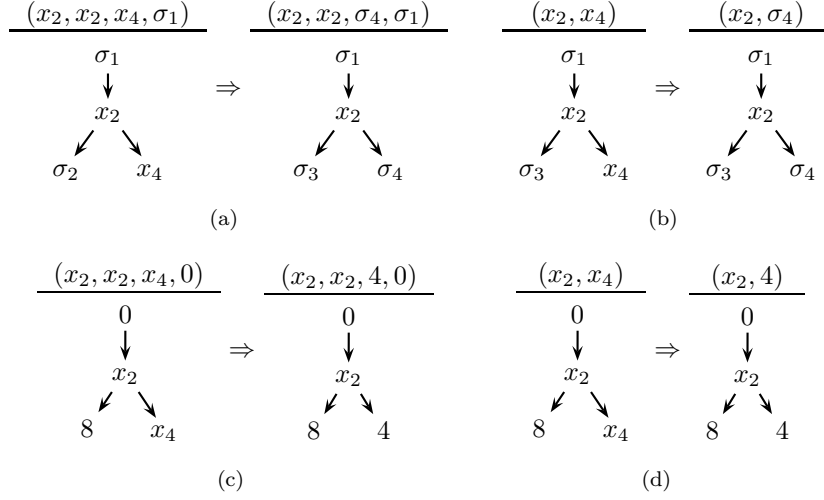


Figure 19: Rule (a) has a non-pure lhs. Rule (b) is the purification of rule (a), and expresses precisely the same information. Rules (c) and (d) are two example instantiations.

P. As an illustration, the lhs of rule (a) of Figure 19 is impure, while the lhs of rule (b) is pure.

Consider the pARs in Figure 19(a) and Figure 19(b), and their instantiations in Figure 19(c) and Figure 19(d). The rules in Figure 19(a) and Figure 19(c) have an impure lhs. If we apply the iARs in Figure 19(c) and Figure 19(d) to the data graph G in Figure 4(a), both have the same frequency, namely 2, and the same confidence, namely 33%. Indeed, since the frequency of a tree query is in fact the frequency of its body, repetitions of distinguished variables in the head and the occurrence of parameters in the head do not change the frequency of a tree query. In fact the pAR in Figure 19(b) is the purification of the pAR in Figure 19(a): the repetition of the distinguished variable x_2 is removed from the heads, and the parameter σ_1 is removed from the heads.

Hence, a pAR with an impure lhs can always be rewritten to an equivalent pAR with a pure lhs, in such a way that all instantiations of the pAR with the impure lhs correspond to instantiations of the pAR with pure lhs, with the same confidence and frequency. Indeed, take a legal pAR $Q_1 \Rightarrow_\rho Q_2$ with Q_1 not pure. We know that Q_1 's head is mapped to Q_2 's head by some ρ -containment mapping. Hence, we can purify Q_1 by removing all parameters and repetitions of distinguished variables from Q_1 's head, sort the head by the order on the variables, and perform the corresponding actions on Q_2 's head as prescribed by the ρ -containment mapping.

We can conclude that it is sufficient to only consider pARs with pure lhs's. The rhs, however, need not be pure; impure rhs's are in fact interesting, as we will demonstrate next.

Impure rhs's Consider the pAR in Figure 20(a). The rhs is impure since x_2 appears twice in the head. The pAR expresses that a sufficient proportion of the matchings of the lhs pattern, are also matchings of the rhs pattern, which

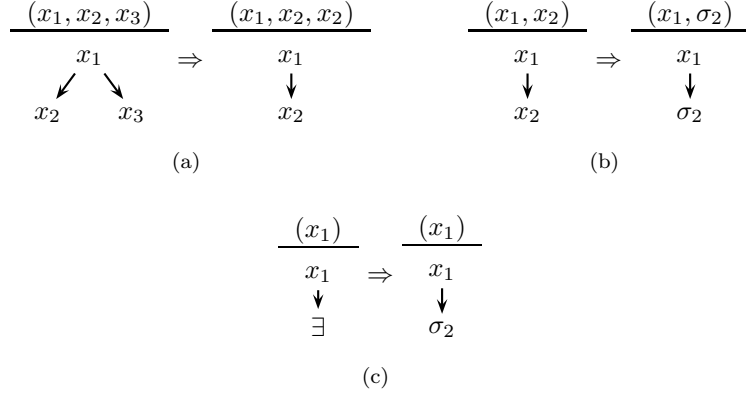


Figure 20: (a) and (b) are pARs with impure rhs. (c) is an ill-advised attempt to purify (b) on the rhs.

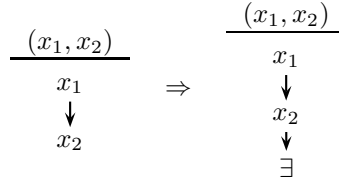


Figure 21: A pAR with a pure rhs.

is the same as the lhs pattern except that x_2 is equal to x_3 . Since the pAR has no parameters, we can identify it with its instantiation by the empty parameter assignment. The confidence is then:

$$\frac{m}{\sum_x \deg^2 x}$$

where m is the number of edges, x ranges over the nodes in the graph, and $\deg x$ is the outdegree of (number of edges leaving) x . Since $m = \sum_x \deg x$, we show by an easy calculation that this confidence is much larger than $1/m$:

$$\begin{aligned} \frac{m}{\sum_x \deg^2 x} &= \frac{\sum_x \deg x}{\sum_x \deg^2 x} \\ &\geq \frac{\sum_x \deg x}{(\sum_x \deg x)^2} \\ &= \frac{1}{\sum_x \deg x} \\ &= \frac{1}{m} \end{aligned}$$

Hence, the sparser the graph (with the number of nodes remaining the same), the higher the confidence, and thus the pAR is interesting in that it tells us something about the sparsity of the graph. As an illustration, on the graph of Figure 4(a) the confidence is 0.4, but on the the graph of Figure 4(b), it is 0.6.

Also consider the pAR in Figure 20(b). Again the rhs is impure since its head contains a parameter. Create an iAR for this pAR with $\alpha = \sigma_2 \mapsto 8$. With confidence c , this iAR then expresses that a fraction of c of all edges point to node 8, which again would be an interesting property of the graph.

The knowledge expressed by the above two example pARs cannot be expressed using pARs with pure rhs's. To illustrate, the pAR of Figure 20(c) may at first seem equivalent (and has a pure rhs) to that of Figure 20(b). On second thought, however, it says nothing about the proportion of *edges* pointing to σ_2 , but only about the proportion of *nodes* with an edge to σ_2 .

Of course, we are not implying that pARs with pure rhs's are uninteresting. But all they can express are statements about the proportion of matchings of the lhs that can be specialized or extended to a matching of the rhs (another example is in Figure 21, which says something about the proportion of edges that can be extended); they cannot say anything about the proportion of matchings of the lhs that satisfy certain equalities in the distinguished variables.

Free Constants Most treatments of conjunctive database queries [9, 2, 40] allow arbitrary constants in the head. In our treatment, a constant can only appear in the head as the value of a parameter. Fortunately this is enough. We do not need to consider “free” constants, i.e., constants not corresponding to a parameter value. To see this, first consider the possibility of free constants in the lhs. The same argument we already gave to assume that the lhs is pure can be used to dismiss this possibility. Next consider a constant in the rhs of an iAR $(Q_1 \Rightarrow_\rho Q_2, \alpha)$, with $Q_1 = (H_1, P_1)$ and $Q_2 = (H_2, P_2)$ and Q_1 already pure. Then there must be a ρ -containment mapping $f : Q_1 \rightarrow Q_2$, with $f(H_1) = H_2$, for the iAR to be legal. Hence, a constant a can only appear in H_2 by one of the following two possibilities:

1. $a = \alpha(f(\sigma)) = \alpha(\rho(\sigma))$, with $\sigma \in H_1$; or
2. $a = \alpha(f(x))$, with x a distinguished variable in H_1 .

However, in both cases a is not actually free, being equal to a parameter value.

5.2 Overall Approach

Given the inputs: G ; $Q_{\text{left}} = (H_{\text{left}}, P_{\text{left}})$; minconf ; and minsup , an outline of our algorithm for the association rule mining problem is that of four nested loops:

1. Generate, incrementally, all possible trees of increasing sizes. Avoid trees that are isomorphic to previously generated ones. The height of the generated trees must be at least the height of the tree underlying P_{left} . (When enough trees have been generated, this loop can be terminated.)
2. For each new generated tree T , generate all frequent instantiated tree patterns P^α based on that tree.

These first two loops are nothing but our algorithm for mining frequent tree queries as presented in Section 4.

3. For each parameterized tree pattern P , generate all containment mappings f from P_{left} to P . Here, a plain “containment mapping” is a ρ -containment mapping, as defined in Section 3.3, for some ρ . Note that ρ then equals $f|_{\Sigma_{\text{left}}}$.
4. For each f , generate the parameterized tree query $Q_{\text{right}} = (f(H_{\text{left}}), P)$, and all parameter assignments α such that $(Q_{\text{left}} \Rightarrow_{\rho} Q_{\text{right}}, \alpha)$ is frequent; and the confidence exceeds *minconf*. The generation of all these α ’s happens in a parallel fashion.

This approach is complete, i.e., it will output everything that must be output. In proof, consider a legal, frequent and confident iAR $(Q_{\text{left}} \Rightarrow_{\rho_0} Q_{\text{right}}, \alpha_0)$, with $Q_{\text{right}} = (H_{\text{right}}, P_{\text{right}})$. The tree T is the underlying tree of P_{right} ; P_{right} is a tree pattern P in loop 2; the containment mapping f in loop 3 is the ρ_0 -containment mapping that exists since the iAR is legal; H_{right} is $f(H_{\text{left}})$; and α in loop 4 is α_0 .

The reader may wonder whether loop 3 cannot be organized in a level-wise fashion. This is not obvious, however, since any two queries of the form $((f_1(H_{\text{left}}), P), \alpha)$ and $((f_2(H_{\text{left}}), P), \alpha)$ have exactly the same frequency, namely that of P^α . Loop 4, however, is levelwise because it is based on loop 2 which is levelwise.

As already mentioned, these first two loops are nothing but our algorithm for mining frequent tree queries as presented in Section 4. As already explained in Section 4.6, in loop 2 we build up a structured database containing all frequency tables for all trees in loop 1. We call this database the *pattern database*. In fact these two loops should be regarded as a preprocessing step; once built up, this pattern database can be used to generate association rules.

Hence, in practice an outline for our rule-mining algorithm is the following:

1. Preprocessing step: Generate a pattern database D using the algorithm discussed in Section 4. Halt this algorithm when enough patterns are generated.
2. Consider, in a levelwise order, each parameterized tree pattern P that has frequent instantiations in D , and such that the height of the underlying tree of P is at least the height of the underlying tree of P_{left} .
3. For each parameterized tree pattern P , generate all containment mappings f from P_{left} to P and let ρ be $f|_{\Sigma_{\text{left}}}$.
4. For each f , generate the parameterized tree query $Q = (f(H_{\text{left}}), P)$, and all parameter assignments α such that $(Q_{\text{left}} \Rightarrow_{\rho} Q, \alpha)$ is frequent; and the confidence exceeds *minconf*. The generation of all these α ’s happens in a parallel fashion.

We present loops 3 and 4 in detail in Sections 5.3 and 5.4. In Section 5.6, we will show how our overall approach must be refined so that the generation of equivalent association rules is avoided.

5.3 Generation of Containment Mappings

In this section, we discuss loop 3, the generation of all containment mappings f from P_{left} to P . So, we need to solve the following problem: Given two

parameterized tree patterns P_1 and P_2 , find all containment mappings f from P_1 to P_2 .

Since the patterns are typically small, a naive algorithm suffices. For a node x_1 of P_1 and a node x_2 of P_2 , we say that x_1 “matches” x_2 if there is a containment mapping f from the subpattern of P_1 rooted at x_1 to the subpattern of P_2 rooted at x_2 such that $f(x_1) = x_2$. In a first phase, we determine for every node y of P_2 separately whether the root r_1 of P_1 matches y . While doing so, we also determine for every other node x_1 of P_1 , and every node x_2 below y at the same distance as x_1 is from r_1 , whether x_1 matches x_2 . We store all these boolean values in a two-dimensional matrix *Map*. The function for filling in *Map* is given in function 4. In line 2 of this function we mean by “ $x_1 \mapsto x_2$ is legal”, that if x_1 is a distinguished variable, then x_2 is a distinguished variable or a parameter; and if x_1 is a parameter then x_2 is a parameter, as prescribed by the definition of a ρ -containment mapping in Section 3.3.

Function 4 Function for filling in *Map*

```

1: bool FillInn( $\mathbf{x_1} \in \mathbf{P_1}, \mathbf{x_2} \in \mathbf{P_2}$ )
2: if  $x_1 \mapsto x_2$  is legal then
3:   Match := true;
4:   for each child  $c_1$  of  $x_1$  from left to right do
5:     MatchChild := false;
6:     for each child  $c_2$  of  $x_2$  from left to right do
7:       MatchChild := MatchChild OR FillInn( $c_1, c_2$ )
8:     end for
9:   Match := Match AND MatchChild;
10: end for
11: Map[ $x_1, x_2$ ] := Match;
12: return Match;
13: else
14:   Map[ $x_1, x_2$ ] := false;
15: return false
16: end if

```

This first phase compares every possible pair (x_1, x_2) , with x_1 a node in P_1 and x_2 a node in P_2 , at most once. Indeed, if x_1 is at distance d from r_1 , then x_1 will be compared to x_2 only during the matching of r_1 with the node y that is d steps above x_2 in P_2 (if existing). We thus have an $O(n_1 \times n_2)$ algorithm, where n_1 (n_2) is the number of nodes in P_1 (P_2).

In a second phase, we output all containment mappings. Initially, by a synchronous preorder traversal of P_1 and P_2 , we map each node of P_1 to the first matching node of P_2 . We store this first mapping in a one-dimensional matrix *Cm*. In function 5 an outline for finding the initial containment mapping is given.

In each subsequent step, we look for the last node x_1 (in preorder) of P_1 , currently matched to some node x_2 , with the property that x_1 can also be matched to a right sibling x_3 of x_2 , and now map x_1 to the first such x_3 . The mappings of all nodes of P_1 coming after x_1 are reinitialized. Every such step takes time that is linear in n_1 and n_2 . Of course, the total number of different containment mappings may well be exponential in n_1 . An outline of this step is given in Function 6.

Function 5 Function for finding the initial containment mapping

```
1: Init( $\mathbf{x}_1 \in \mathbf{P}_1, \mathbf{x}_2 \in \mathbf{P}_2$ )
2:  $\text{Cm}[x_1] := x_2$ ;
3: for each child  $c_1$  of  $x_1$  from left to right do
4:   for each child  $c_2$  of  $x_2$  from left to right do
5:     if  $\text{Map}[c_1, c_2]$  then
6:        $\text{Init}(c_1, c_2)$ ;
7:       Break;
8:     end if
9:   end for
10: end for
```

Function 6 Function for finding the other containment mappings

```
1: bool Step( $\mathbf{x} \in \mathbf{P}_1$ )
2:  $\text{Found} := \text{false}$ ;
3: for each child  $c$  from  $x$  from right to left do
4:   if  $\text{Step}(c)$  then
5:      $\text{Found} := \text{true}$ ;
6:     Break;
7:   end if
8: end for
9: if  $\text{Found}$  then
10:  for each right-sibling  $z$  of  $c$  from left to right do
11:     $p_2 := \text{Cm}[x]$ ;
12:    for each child  $c_2$  of  $p_2$  from left to right do
13:      if  $\text{Map}[z, c_2]$  then
14:         $\text{Init}(z, c_2)$ 
15:      end if
16:    end for
17:  end for
18:  return true;
19: else
20:  if  $x$  is the root of  $P_1$  then
21:    return false;
22:  else
23:     $m := \text{Cm}[x]$ ;
24:    for each right-sibling  $s$  of  $m$  from left to right do
25:      if  $\text{Map}[x, s]$  then
26:         $\text{Init}(x, s)$ 
27:        Break;
28:      end if
29:    end for
30:    return true;
31:  end if
32: end if
```

The complete outline for the generation of all containment mappings is given in Function 7.

Function 7 Function for generating all containment mappings from P_1 to P_2

```

1: GenerateCm( $P_1, P_2$ )
2: Initialize Map;
3:  $r_1 := \text{root of } P_1$ ;
4: for each  $x_2 \in P_2$  in preorder do
5:   FillIn( $r_1, x_2$ );
6: end for
7: for each node  $x_2 \in P_2$  in preorder do
8:   if Map[ $r_1, x_2$ ] then
9:     Initialize Cm;
10:    Init( $r_1, x_2$ )
11:    repeat
12:      Output Cm;
13:    until not Step( $r_1$ )
14:   end if
15: end for

```

We can thus easily generate all containment mappings f from P_{left} to P as required for loop 3 of our overall algorithm. Note, however, that in loop 4 these mappings are used to produce the head $f(H_{\text{left}})$ of query Q_{right} . For Q_{right} to be a legal query, this head must contain all distinguished variables of P . Hence, we only pass to loop 4 those f whose image contains all distinguished variables of P .

5.4 Generation of Parameter Assignments

In loop 4, our task is the following. Given a containment mapping $f : P_{\text{left}} \rightarrow P$, let $\rho = f|_{\Sigma_{\text{left}}}$, and generate all parameter assignments α such that $(Q_{\text{left}} \Rightarrow_{\rho} (f(H_{\text{left}}), P), \alpha)$ is frequent and confident in G . We show how this can be done in a parallel database-oriented fashion.

Recall from Section 4.6 that the frequency tables for P_{left} and P are available in a relational database. Our crucial observation is that we can compute precisely the required set of parameter assignments α , together with the frequency and confidence of the corresponding association rules, by a single relational algebra expression. This expression has the following form:

$$\pi_{plist} \sigma_{\frac{FreqTab_P.freq}{FreqTab_{P_{\text{left}}}.freq} \geq minconf} (FreqTab_{P_{\text{left}}} \bowtie_{\theta} FreqTab_P)$$

Here, π denotes projection, σ denotes selection, and \bowtie denotes join. The join condition θ and the projection list $plist$ are defined as follows. For θ , we take the conjunction:

$$\bigwedge_{\sigma \in \Sigma_{\text{left}}} FreqTab_{P_{\text{left}}}.\sigma = FreqTab_P.\rho(\sigma)$$

Furthermore, $plist$ consists of all attributes $P_{\text{left}}.\sigma_{\text{left}}$, with $\sigma_{\text{left}} \in \Sigma_{\text{left}}$; all attributes $P.\sigma$, with $\sigma \in \Sigma$; together with the attributes $FreqTab_P.freq$ and $FreqTab_P.freq / FreqTab_{P_{\text{left}}}.freq$.

Referring back to our overall algorithm (Section 5.2), we thus generate, for each pattern P from loop 2 and each containment mapping f in loop 3, all association rules with the given Q_{left} as lhs in parallel, by one relational database query (which can be implemented by a simple SQL select-statement).

Example. Consider Q_{left} and $P = (\Pi, \Sigma)$ as shown in Figures 22(a) and 22(b). We have $\Sigma_{\text{left}} = \{\sigma_1, \sigma_4\}$ and $\Pi_{\text{left}} = \{x_3, x_6\}$, and $\Sigma = \{\sigma_1, \sigma_4, \sigma_5\}$ and $\Pi = \{x_3\}$. Take the following containment mapping f from P_{left} to P :

f	
σ_1	σ_1
x_1	x_2
\exists_3	\exists
σ_4	σ_4
x_5	x_2
\exists_6	\exists
x_7	σ_4

Then the rhs query Q_{right} equals $((x_2, x_2, \sigma_4), P)$, and the relational algebra expression for computing all parameter assignments and their corresponding frequencies and confidences looks as follows:

$$\pi_{plist} \sigma_{\frac{FreqTab_P \cdot freq}{FreqTab_{P_{\text{left}}} \cdot freq} \geq minconf} (FreqTab_{P_{\text{left}}} \bowtie_{\theta} FreqTab_P)$$

with $plist$ equal to

$$FreqTab_{P_{\text{left}}} \cdot \sigma_1, FreqTab_{P_{\text{left}}} \cdot \sigma_4, FreqTab_P \cdot \sigma_1, FreqTab_P \cdot \sigma_4, FreqTab_P \cdot \sigma_5, \\ FreqTab_P \cdot freq, FreqTab_P \cdot freq / FreqTab_{P_{\text{left}}} \cdot freq$$

and θ equal to

$$FreqTab_P \cdot \sigma_1 = FreqTab_{P_{\text{left}}} \cdot \sigma_1 \wedge FreqTab_P \cdot \sigma_4 = FreqTab_{P_{\text{left}}} \cdot \sigma_4$$

In SQL, we get:

```
SELECT freqQleft.x1, freqQleft.x4, freqP.x1, freqP.x4,
       freqP.x5, freqP.freq, freqP.freq/freqQleft.freq
FROM freqP, freqQleft
WHERE freqQleft.x1= freqP.x1 AND freqQleft.x4=freqP.x4
      AND freqP.freq/freqQleft.freq >= minconf
```

5.5 Example Run

In this Section we give an example run of the algorithm discussed in Section 5. We use the same data graph G , unordered rooted tree T , and minimum support threshold, 3, as in the example run in Section 4.4.4. The fixed lhs tree query is given in Figure 23(a), its corresponding frequency table in Figure 23(b), and the minimum confidence threshold is 30%. All frequent tree patterns based on T were already generated in the example run of Section 4.5.5.

The example run then looks as follows:

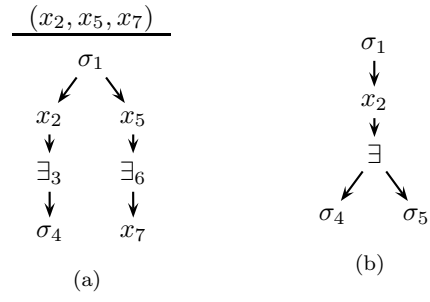


Figure 22: Example Q_{left} and P .

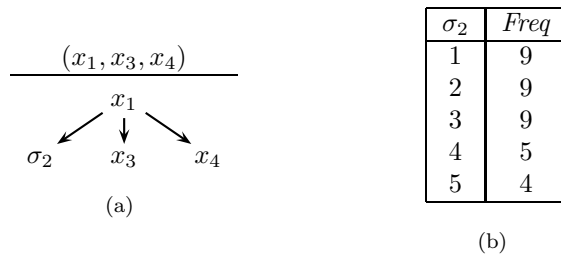
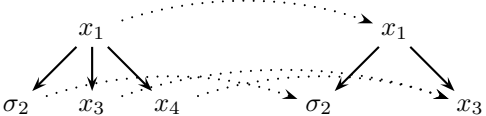
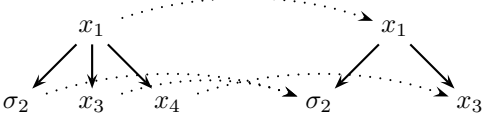
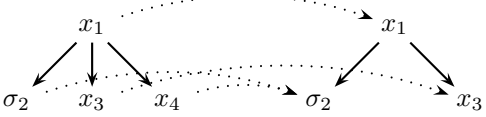


Figure 23: The fixed lhs and its frequency table for the example run in Section 5.5.

P	Containment Mapping		Q_{right}	$ConfTab$																			
Level 0																							
(\emptyset, \emptyset)	No Containment Mappings																						
Level 1																							
$(\emptyset, \{x_1\})$	No Containment Mappings																						
$(\emptyset, \{x_2\})$		$\frac{(x_1, x_3, x_3)}{x_1 \rightarrow \sigma_2 \quad x_1 \rightarrow x_3}$	<table><tr><th>$P_{\text{left}.\sigma_2}$</th><th>$P.\sigma_2$</th><th>$Freq$</th><th>$Conf$</th></tr><tr><td>1</td><td>1</td><td>3</td><td>33%</td></tr><tr><td>2</td><td>2</td><td>3</td><td>33%</td></tr><tr><td>3</td><td>3</td><td>3</td><td>33%</td></tr><tr><td>4</td><td>4</td><td>3</td><td>60%</td></tr></table>	$P_{\text{left}.\sigma_2}$	$P.\sigma_2$	$Freq$	$Conf$	1	1	3	33%	2	2	3	33%	3	3	3	33%	4	4	3	60%
$P_{\text{left}.\sigma_2}$	$P.\sigma_2$	$Freq$	$Conf$																				
1	1	3	33%																				
2	2	3	33%																				
3	3	3	33%																				
4	4	3	60%																				
$(\emptyset, \{x_2\})$		$\frac{(x_1, \sigma_2, x_3)}{x_1 \rightarrow \sigma_2 \quad x_1 \rightarrow x_3}$	<table><tr><th>$P_{\text{left}.\sigma_2}$</th><th>$P.\sigma_2$</th><th>$Freq$</th><th>$Conf$</th></tr><tr><td>1</td><td>1</td><td>3</td><td>33%</td></tr><tr><td>2</td><td>2</td><td>3</td><td>33%</td></tr><tr><td>3</td><td>3</td><td>3</td><td>33%</td></tr><tr><td>4</td><td>4</td><td>3</td><td>60%</td></tr></table>	$P_{\text{left}.\sigma_2}$	$P.\sigma_2$	$Freq$	$Conf$	1	1	3	33%	2	2	3	33%	3	3	3	33%	4	4	3	60%
$P_{\text{left}.\sigma_2}$	$P.\sigma_2$	$Freq$	$Conf$																				
1	1	3	33%																				
2	2	3	33%																				
3	3	3	33%																				
4	4	3	60%																				
$(\emptyset, \{x_2\})$		$\frac{(x_1, x_3, \sigma_2)}{x_1 \rightarrow \sigma_2 \quad x_1 \rightarrow x_3}$	<table><tr><th>$P_{\text{left}.\sigma_2}$</th><th>$P.\sigma_2$</th><th>$Freq$</th><th>$Conf$</th></tr><tr><td>1</td><td>1</td><td>3</td><td>33%</td></tr><tr><td>2</td><td>2</td><td>3</td><td>33%</td></tr><tr><td>3</td><td>3</td><td>3</td><td>33%</td></tr><tr><td>4</td><td>4</td><td>3</td><td>60%</td></tr></table>	$P_{\text{left}.\sigma_2}$	$P.\sigma_2$	$Freq$	$Conf$	1	1	3	33%	2	2	3	33%	3	3	3	33%	4	4	3	60%
$P_{\text{left}.\sigma_2}$	$P.\sigma_2$	$Freq$	$Conf$																				
1	1	3	33%																				
2	2	3	33%																				
3	3	3	33%																				
4	4	3	60%																				
$(\{x_1\}, \emptyset)$	No containment mappings																						
Level 2																							

$(\emptyset, \{x_1, x_2\})$		$\frac{(\sigma_1, x_3, x_3)}{\sigma_1}$	<table><tr><th>$P_{\text{left}.\sigma_2}$</th><th>$P.\sigma_1$</th><th>$P.\sigma_2$</th><th>$Freq$</th><th>$Conf$</th></tr><tr><td>0</td><td>1</td><td>1</td><td>3</td><td>33%</td></tr><tr><td>0</td><td>2</td><td>2</td><td>3</td><td>33%</td></tr><tr><td>0</td><td>3</td><td>3</td><td>3</td><td>33%</td></tr></table>	$P_{\text{left}.\sigma_2}$	$P.\sigma_1$	$P.\sigma_2$	$Freq$	$Conf$	0	1	1	3	33%	0	2	2	3	33%	0	3	3	3	33%
$P_{\text{left}.\sigma_2}$	$P.\sigma_1$	$P.\sigma_2$	$Freq$	$Conf$																			
0	1	1	3	33%																			
0	2	2	3	33%																			
0	3	3	3	33%																			
$(\emptyset, \{x_1, x_2\})$		$\frac{(\sigma_1, \sigma_2, x_3)}{\sigma_1}$	<table><tr><th>$P_{\text{left}.\sigma_2}$</th><th>$P.\sigma_1$</th><th>$P.\sigma_2$</th><th>$Freq$</th><th>$Conf$</th></tr><tr><td>0</td><td>1</td><td>1</td><td>3</td><td>33%</td></tr><tr><td>0</td><td>2</td><td>2</td><td>3</td><td>33%</td></tr><tr><td>0</td><td>3</td><td>3</td><td>3</td><td>33%</td></tr></table>	$P_{\text{left}.\sigma_2}$	$P.\sigma_1$	$P.\sigma_2$	$Freq$	$Conf$	0	1	1	3	33%	0	2	2	3	33%	0	3	3	3	33%
$P_{\text{left}.\sigma_2}$	$P.\sigma_1$	$P.\sigma_2$	$Freq$	$Conf$																			
0	1	1	3	33%																			
0	2	2	3	33%																			
0	3	3	3	33%																			
$(\emptyset, \{x_1, x_2\})$		$\frac{(\sigma_1, x_3, \sigma_2)}{\sigma_1}$	<table><tr><th>$P_{\text{left}.\sigma_2}$</th><th>$P.\sigma_1$</th><th>$P.\sigma_2$</th><th>$Freq$</th><th>$Conf$</th></tr><tr><td>0</td><td>1</td><td>1</td><td>3</td><td>33%</td></tr><tr><td>0</td><td>2</td><td>2</td><td>3</td><td>33%</td></tr><tr><td>0</td><td>3</td><td>3</td><td>3</td><td>33%</td></tr></table>	$P_{\text{left}.\sigma_2}$	$P.\sigma_1$	$P.\sigma_2$	$Freq$	$Conf$	0	1	1	3	33%	0	2	2	3	33%	0	3	3	3	33%
$P_{\text{left}.\sigma_2}$	$P.\sigma_1$	$P.\sigma_2$	$Freq$	$Conf$																			
0	1	1	3	33%																			
0	2	2	3	33%																			
0	3	3	3	33%																			
$(\{x_1\}, \{x_2\})$	No containment mappings																						
Level 3																							
$(\{x_1\}, \{x_2, x_3\})$	No containment mappings																						

5.6 Equivalent Association Rules

In this section, we make a number of modifications to the algorithm described so far, so as to avoid duplicate work on equivalent rules.

Let us first look at an example of the duplicate work that the algorithm presented until now performs. Consider Q_{left} , $Q_1 = (f_1(H_{\text{left}}), P)$, $Q_2 = (f_2(H_{\text{left}}), P)$; and $Q_3 = (f_3(H_{\text{left}}), P)$ in Figure 24 with f_1 , f_2 and f_3 as follows:

f_1		f_2		f_3	
x_1	u_1	x_1	u_1	x_1	u_1
x_2	u_2	x_2	u_3	x_2	u_2
x_3	u_2	x_3	u_2	x_3	u_3
x_4	u_3	x_4	u_2	x_4	u_3

Furthermore, consider pAR1: $Q_{\text{left}} \Rightarrow Q_1$; pAR2: $Q_{\text{left}} \Rightarrow Q_2$ and pAR3: $Q_{\text{left}} \Rightarrow Q_3$.

The confidence of the first rule (pAR1) equals the proportion of tuples from the answer set of Q_{left} where the values for variables x_2 and x_3 are equal (in the rhs those equal variables are represented by variable u_2 , and the lhs variable x_4 is represented by the rhs variable u_3). Similarly, the confidence of the second rule (pAR2) equals the proportion of tuples from the answer set of Q_{left} where the values for the variables x_3 and x_4 are equal (again the equal lhs variables x_3 and x_4 are represented by the rhs variable u_2 , and the lhs variable x_2 is represented by the rhs variable u_3). Since, due to the symmetry in the lhs pattern, the columns for x_2 , x_3 and x_4 are fully interchangeable in the answer set of Q_{left} , both rules convey precisely the same information: their confidences are equal. The third rule (pAR3) is yet another representation of the same association, but now the equal lhs variables x_3 and x_4 are represented by the rhs variable u_3 . Again, it has the same confidence as pAR1 and pAR2.

It is important to note that the above pARs only differ in the containment mappings f_1 , f_2 and f_3 that generate the rhs head. The algorithm discussed until now generates all these pARs, since we do not perform any check on the containment mappings generated in loop 3 of the overall approach (Section 5.2).

In this Subsection, motivated by the above example, we consider the general problem of when two pARs $Q_{\text{left}} \Rightarrow_{\rho_1} Q_1$ and $Q_{\text{left}} \Rightarrow_{\rho_2} Q_2$ are equivalent, where Q_1 and Q_2 are of the form $(f_1(H_{\text{left}}), P)$ and $(f_2(H_{\text{left}}), P)$ for some common rhs pattern P , and containment mappings f_1 and f_2 from P_{left} to P . (Thus ρ_1 is $f_1|_{\Sigma_{\text{left}}}$ and ρ_2 is $f_2|_{\Sigma_{\text{left}}}$.) Since such two pARs differ only in f_1 and f_2 we can actually focus on f_1 and f_2 .

It is important to remember for the rest of this Subsection that P_{left} and P are arbitrary but fixed. Furthermore, without loss of generality we assume that the nodes of P_{left} and P are disjoint. This assumption greatly simplifies the representation of containment mappings by graphs, as we will see shortly.

Equivalent Containment Mappings Recall from Section 4.5.3 that an *isomorphism* from a parameterized tree pattern P_1 to a parameterized tree pattern P_2 is a homomorphism from P_1 to P_2 that is a bijection and that maps distinguished nodes to distinguished nodes, parameters to parameters and existential nodes to existential nodes. We now formalize equivalent containment mappings as follows: Two containment mappings f_1 and f_2 are *equivalent* if the structures

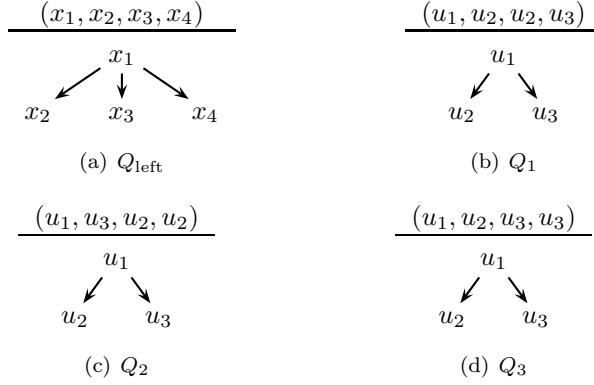


Figure 24: Queries to illustrate the duplicate work in the association mining algorithm

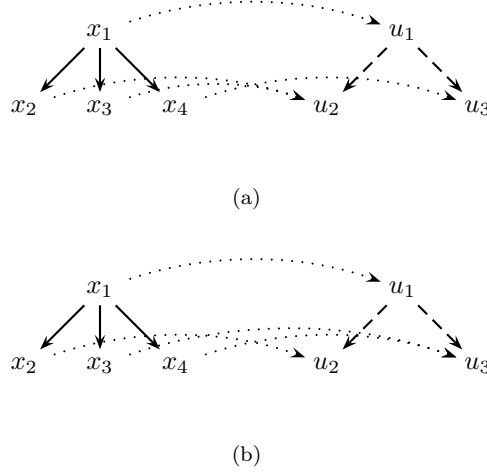


Figure 25: The graph representations of f_1 and f_3 .

$(P_{\text{left}}, P, f_1)$ and $(P_{\text{left}}, P, f_2)$ are isomorphic. Specifically, there must exist isomorphisms (actually automorphisms) $g : P_{\text{left}} \rightarrow P_{\text{left}}$ and $h : P \rightarrow P$ such that $f_2 \circ g = h \circ f_1$.

Consider for instance f_1 and f_3 from the example above, then h swaps u_2 and u_3 , and g is the cyclic permutation $u_2 \mapsto u_3 \mapsto u_4 \mapsto u_2$.

5.6.1 Testing for equivalence

To test for equivalent containment mappings efficiently, we represent them using graphs.

Graph representation of a containment mapping The graph representation of a containment mapping $f : P_{\text{left}} \rightarrow P$ is a directed, edge- and vertex-colored graph, with set of vertices $V_f = \text{Vertices}(P_{\text{left}}) \cup \text{Vertices}(P)$ and set of

edges $E_f = \text{Edges}(P_{\text{left}}) \cup \text{Edges}(P) \cup \{(v, w) \mid f(v) = w\}$ (with the understanding that the edges of P_{left} and P go from parent to child). We use different colors for the edges of P_{left} , the edges of P and the pairs in f , and we also use different colors for the distinguished nodes, the existential nodes and the parameters.

As an illustration, Figure 25 shows the graph representation of f_1 and f_3 from our example in the introduction above.

Graph Isomorphism Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *colored isomorphic* if there exists a bijection $\varphi : V_1 \rightarrow V_2$, extended to edges $(v, w) \in E_1$ in a natural way by $\varphi(v, w) = (\varphi(v), \varphi(w))$, such that the colors of vertices and edges are preserved by φ , and such that $(v, w) \in E_1 \Leftrightarrow (\varphi(v), \varphi(w)) \in E_2$.

The following Lemma shows then the utility of the colored graph representation of containment mappings.

Lemma 6. *Two containment mappings are equivalent if and only if their colored graph representations are isomorphic.*

Proof. Let us start with the only-if direction. Consider two equivalent containment mappings f_1, f_2 from P_{left} to P . By definition of equivalent containment mappings, there exist isomorphisms $g : P_{\text{left}} \rightarrow P_{\text{left}}$ and $h : P \rightarrow P$ such that $f_2 \circ g = h \circ f_1$. Now take $\varphi = g \cup h$. Then, φ is clearly a bijection from V_{f_1} to V_{f_2} , and clearly preserves the colors of vertices and edges of G_{f_1} . Let $(v, w) \in E_{f_1}$. We show that φ is indeed an isomorphism from G_{f_1} to G_{f_2} . There are three possibilities:

1. $(v, w) \in \text{Edges}(P_{\text{left}})$. Note that then also $g(v, w) \in \text{Edges}(P_{\text{left}})$. We have:

$$\begin{aligned}
 (v, w) \in E_{f_1} &\Leftrightarrow (v, w) \in \text{Edges}(P_{\text{left}}) \\
 &\quad g \text{ is an automorphism in } P_{\text{left}} \\
 &\Leftrightarrow g(v, w) \in \text{Edges}(P_{\text{left}}) \\
 &\quad \varphi(v, w) = g(v, w) \\
 &\Leftrightarrow g(v, w) \in E_{f_2} \\
 &\quad \varphi(v, w) \in E_{f_2} \\
 &\Leftrightarrow \varphi(v, w) \in E_{f_2}
 \end{aligned}$$

2. $(v, w) \in \text{Edges}(P)$. Note that then also $h(v, w) \in \text{Edges}(P)$. We have:

$$\begin{aligned}
 (v, w) \in E_{f_1} &\Leftrightarrow (v, w) \in \text{Edges}(P) \\
 &\quad h \text{ is an automorphism in } P \\
 &\Leftrightarrow h(v, w) \in \text{Edges}(P) \\
 &\quad \varphi(v, w) = h(v, w) \\
 &\Leftrightarrow h(v, w) \in E_{f_2} \\
 &\quad \varphi(v, w) \in E_{f_2} \\
 &\Leftrightarrow \varphi(v, w) \in E_{f_2}
 \end{aligned}$$

3. $w = f_1(v)$. We have:

$$\begin{aligned}
(v, w) \in E_{f_1} &\Leftrightarrow v = f_1(w) \\
&\Leftrightarrow h(v) = h(f_1(w)) \\
&\Leftrightarrow h(v) = f_2(g(w)) \\
&\Leftrightarrow \varphi(v) = f_2(\varphi(w)) \\
&\Leftrightarrow (\varphi(v), \varphi(w)) \in E_{f_2}
\end{aligned}$$

So we can conclude that G_{f_1} and G_{f_2} are indeed colored isomorphic.

Let us now look at the if-direction. Let φ be the given isomorphism from G_{f_1} to G_{f_2} . Now take $g = \varphi|_{\text{Vertices}(P_{\text{left}})}$ and $h = \varphi|_{\text{Vertices}(P)}$. To prove that f_1 and f_2 are equivalent it suffices to show that:

1. g is an isomorphism from P_{left} to P_{left} ;
2. h is an isomorphism from P to P ; and
3. $f_2 \circ g = h \circ f_1$.

Items 1 and 2 hold because φ preserves the colors. For 3, let $v \in P_{\text{left}}$. Since φ is a graph isomorphism $f_2(\varphi(v))$ equals $\varphi(f_1(v))$. We then have:

$$\begin{aligned}
f_2(g(v)) &= f_2(\varphi(v)) \\
&= \varphi(f_1(v)) \\
&= h(f_1(v))
\end{aligned}$$

□

So, using graph isomorphism (to be precise, edge and vertex colored directed graph isomorphism), we can test for equivalence. Since our patterns are not very large, fast heuristics for graph isomorphism can be used. We use the program Nauty [32, 33], which is considered as the fastest heuristic for graph isomorphism. Nauty is very efficient for small, dense random graphs [15]. Since our graph representations are typically small (no more than 20 vertices) and dense, this works well in our case.

Theoretically this situation is not entirely satisfying, as graph isomorphism is not known to be efficiently (polynomial-time) solvable in general. We can show however that equivalence of our containment mappings is really as hard as the general graph isomorphism problem. This hardness argument is presented in the following Section 5.6.2. As special case of the equivalence problem that is solvable in polynomial time is presented in Section 5.6.3

5.6.2 Hardness argument

First recall from graph theory that a graph $B = (V, E)$ is *bipartite* if V can be split in two disjoint parts, $V = V^a \cup V^b$ with $V^a \cap V^b = \emptyset$, such that for each $(v, w) \in E$ then $v \in V^a$ and $w \in V^b$. The vertices in V^a are called lhs vertices and those in V^b rhs vertices (left-hand side, right-hand side).

We first reduce the problem of bipartite graph isomorphism to equivalence of our containment mapping. Let $B_1 = (V_1, E_1)$ and $B_2 = (V_2, E_2)$ be bipartite graphs. We describe an efficient construction that produces from B_1 and B_2

two association rules $(P_{\text{left}}, P, f_1)$ and $(P_{\text{left}}, P, f_2)$ such that B_1 and B_2 are isomorphic if and only if the association rules are equivalent. This construction reduces the bipartite graph isomorphism problem to equivalence of containment mappings.

Without loss of generality, we assume that B_1 and B_2 have precisely the same multiset of outdegrees (for vertices of V_1^a and V_2^a), and precisely the same number of vertices in V_1^b and V_2^b . Indeed, if these conditions are not satisfied, then B_1 and B_2 are never isomorphic and our reduction can output some arbitrary P_{left} , P , f_1 and f_2 as long as $(P_{\text{left}}, P, f_1)$ and $(P_{\text{left}}, P, f_2)$ are not equivalent.

The construction is now as follows. By the premisses on B_1 and B_2 , we may assume, without loss of generality, that $V_1^a = V_2^a$ and $V_1^b = V_2^b$. This can be accomplished by sorting the lhs vertices in each graph on their outdegrees and then numbering them arbitrarily (the rhs vertices can simply be numbered arbitrarily).

1. Construction of P_{left} . This is a tree with root called r_{left} and as children of the root, all lhs vertices. Moreover, each lhs vertex v has its own children as follows: if v has outdegree o , then v has o children denoted by $[v, 1]$, $[v, 2]$, ..., $[v, o]$.
2. Construction of P . This is a tree with root called r_{right} , and exactly one child of the root, called c . Moreover, c has as children precisely all rhs vertices.
3. Construction of f_1 . We define $f(r_{\text{left}}) := r_{\text{right}}$, and define $f_1(v) := c$ for each lhs vertex v . Now for each such v , and all outgoing edges $(v, w_1), (v, w_2), \dots, (v, w_o)$ in B_1 , listed in some arbitrary order, we define $f_1([v, i]) := w_i$, for $i = 1, 2, \dots, o$.
4. The construction of f_2 is analogous to that of f_1 , but now we look at the outgoing edges in B_2 .

The construction is illustrated in Figure 26 for two bipartite graphs B_1 and B_2 .

We now show the correctness of our reduction.

Lemma 7. *B_1 and B_2 are isomorphic if and only if $(P_{\text{left}}, P, f_1)$ and $(P_{\text{left}}, P, f_2)$ are isomorphic.*

Proof. For the only-if direction, let ψ be an isomorphism from B_1 to B_2 . We define an isomorphism from $(P_{\text{left}}, P, f_1)$ to $(P_{\text{left}}, P, f_2)$ as follows:

- $\varphi(r_{\text{left}}) = r_{\text{left}}$, $\varphi(r_{\text{right}}) = r_{\text{right}}$ and $\varphi(c) = c$;
- $\varphi(v) = \psi(v)$, for any vertex of B_1 ;
- for any lhs vertex v of outdegree o , and any $i = 1, 2, \dots, o$, let w be the rhs vertex such that $f_1([v, i]) = w$. Then we define $\varphi([v, i]) := [\psi(v), j]$, where j is such that $f_2([\psi(v), j]) = \psi(w)$.

To verify that φ is indeed an isomorphism, we only check that $u = f_1([v, i]) \Leftrightarrow \psi(u) = f_2(\psi([v, i]))$. If $u = f_1([v, i])$ then (v, u) is an edge in B_1 and thus $(\varphi(v), \varphi(u)) = (\psi(v), \psi(w))$ is an edge in B_2 . Hence there exists a j such that,

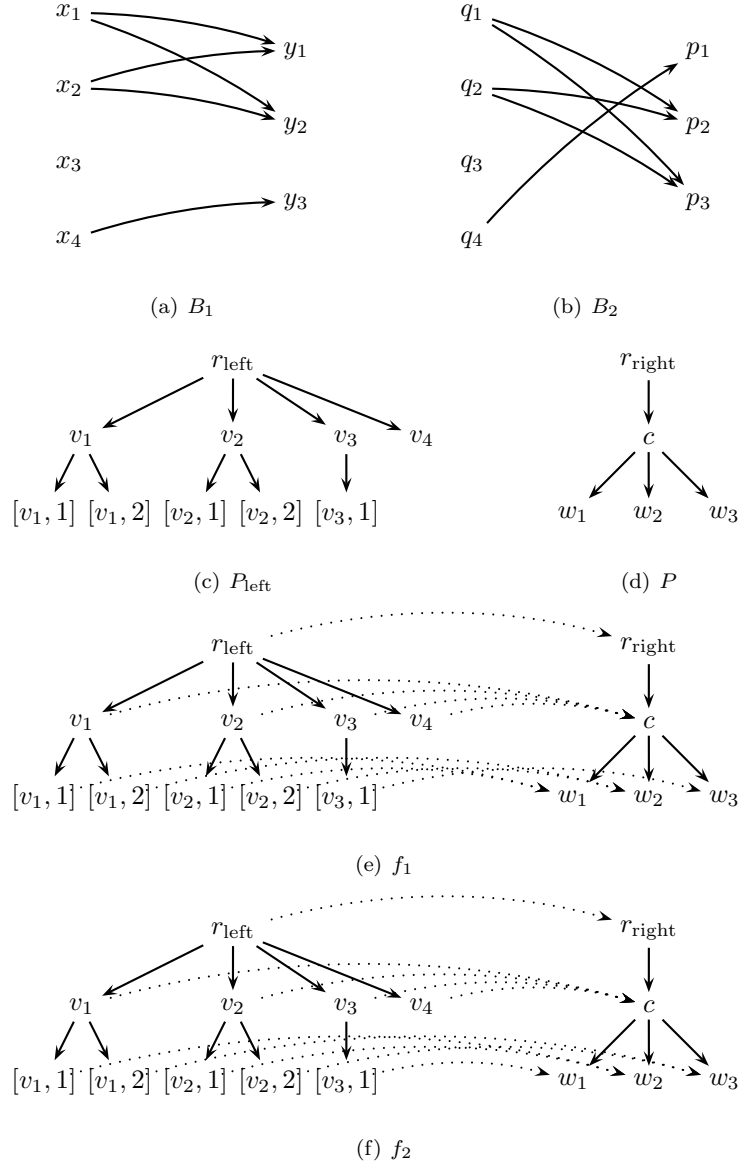


Figure 26: Illustration of the construction of pARs from bipartite graphs.

$\varphi(u) = f_2([\varphi(v), j])$, or equivalent, $\psi(u) = f_2([\psi(v), j])$. By definition of φ we have $\varphi([v, i]) = [\psi(v), j]$ and thus $\varphi(u) = f_2(\varphi([v, i]))$ as desired. Conversely, suppose $\varphi(u) = f_2(\varphi([v, i]))$. By definition of φ , we have $\varphi([v, i])$ equals $[\psi(v), j]$ for some unique j , and $f_2([\psi(v), j])$ equals $\psi(f_1([v, i]))$. Hence, $\psi(u) = \varphi(u) = \psi(f_1([v, i]))$, and thus $u = f_1([v, i])$ as desired.

For the if-direction, let φ be an isomorphism from $(P_{\text{left}}, P, f_1)$ to $(P_{\text{left}}, P, f_2)$. We define an isomorphism ψ from B_1 to B_2 as follows. Actually, ψ is simple φ restricted to the vertices of B_1 . Indeed,

$$\begin{aligned} (v, w) \in E_1 &\Leftrightarrow \exists i : f_1([v, i]) = w \\ &\Leftrightarrow \exists i : f_2(\varphi([v, i])) = \varphi(w) \\ &\Leftrightarrow \exists j : f_2(\varphi(v), j) = \varphi(w) \\ &\Leftrightarrow (\varphi(v), \varphi(w)) \in E_2 \end{aligned}$$

□

We can already conclude from this reduction that equivalence of pARs is really as hard as isomorphism of bipartite directed graphs. The latter problem, however, is well-known to be as hard as isomorphism of general directed graphs. Indeed, any directed graph $G = (V, E)$ can be transformed into the bipartite directed graph $B(G) := (V \cup E, \{(v, (v, w)) \mid (v, w) \in E\} \cup \{((v, w), w) \mid (v, w) \in E\})$, and it is easily verified that G_1 and G_2 are isomorphic if and only if $B(G_1)$ and $B(G_2)$ are isomorphic.

So, we can now conclude that equivalence of our pARs is really as hard as the general graph isomorphism problem. But as we show next, we can still capture an important special case in polynomial time, so that the general graph isomorphism heuristics only have to be applied on instances not captured by the special case.

5.6.3 Polynomial case

The special efficient case is to check whether $(P_{\text{left}}, P, f_1)$ and $(P_{\text{left}}, P, f_2)$ are already isomorphic with g the identity, i.e., whether the structures (P, f_1) and (P, f_2) are already isomorphic. So, we look for an automorphism h of P such that $f_2 = h \circ f_1$. This can be solved efficiently by a reduction to node-labeled tree isomorphism. As explained in Section 4.4, if we know the tree T underlying P , then P is characterized by the pair (Π, Σ) , and thus (P, f) is characterized by (Π, Σ, f) . We can view this triple as a labeling of T , as follows. We label every node y of P with a triple $(b_\Pi, b_\Sigma, f^{-1}(y))$, where b_Π is a bit that is 1 iff $y \in \Pi$; b_Σ is a bit that is defined likewise; and $f^{-1}(y)$ is the set of nodes of P_{left} that are mapped by f to y . Then (P, f_1) and (P, f_2) are isomorphic if and only if the corresponding node-labeled trees are isomorphic, and the latter can be checked in linear time using canonical ordering [4, 10].

5.6.4 The Algorithm

We are now in a position to describe how our general algorithm must be modified to avoid equivalent association rules. There is only extra checking to be done in loop 3 (recall Sections 5.2 and 5.3). For each new containment mapping f from P_{left} to P , we canonize the corresponding node-labeled tree and we check if

the canonical form is identical to an earlier generated canonical form; if so, f is dismissed. We can keep track of the canonical forms seen so far efficiently using a trie data structure. If the canonical form was not yet seen, we can either let f through to loop 4, if the presence of duplicates in the output is tolerable for the application at hand, or we can perform the colored graph isomorphism check of Section 5.6.1 with the containment mappings previously seen, to be absolutely sure we will not generate a duplicate.

6 Certhia: Pattern and Association Browsing

In this Section we introduce an interactive tool, called *Certhia*, for browsing the frequent tree patterns, and generating association rules.

As already noted in Section 4.6, the result of our tree query mining algorithm in Section 4 is a structured database, called a Pattern Database, containing all frequency tables for each tree T that was investigated. This pattern database is an ideal platform for an interactive tool for browsing the frequent queries. However, this pattern database is also an ideal platform for generating association rules as explained in Section 5.2, since the first two loops of association rule algorithm are exactly our tree query mining algorithm.

In a typical scenario for Certhia, the user draws a tree shape, marks some nodes as existential, marks some others as parameters, instantiates some parameters by constants, but possibly also leaves some parameters open. The browser then returns, by consulting the appropriate frequency table in the database, all instantiations of the free parameters that make the pattern frequent, together with the frequency. The user can then select one of these instantiations, set a minconf value, and ask the browser to return all rhs's that form a confident association with the selected pure tree query as lhs. In another scenario the user lets the browser suggest some frequent tree patterns to choose from as an lhs.

Some screenshots of Certhia are given in Figure 27, Figure 28 and Figure 29.

- In Figure 27, the user draws a tree, marks some nodes as existential, some others as parameters, instantiates some parameters with constants, and asks the browser to return all possible instantiations of the remaining parameters and the corresponding frequencies.
- In Figure 28, the user asks the browser to return all association rules for a fixed lhs. The user selects a rhs in the dialog box and asks the browser to return the instantiations and the corresponding frequencies.
- In Figure 29, the browser suggests some frequent tree patterns where the user can choose from.

Efficiency The preprocessing step, i.e., the building up of the pattern database with frequent tree patterns, is of course a hugely intensive task. First because the large datagraph must be accessed intensively, and secondly because the number of frequent patterns is huge. In Section 7.2 we show that this preprocessing step can be implemented with satisfactory performance. Also, in scientific discovery applications it is no problem, indeed typical, if a preprocessing step takes

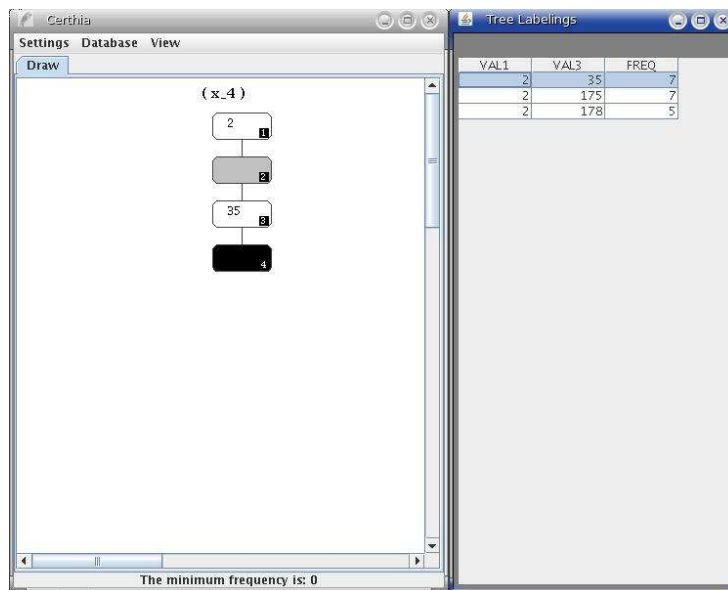


Figure 27: Screenshot of Certhia: the user draws a pattern and asks for the instantiations.

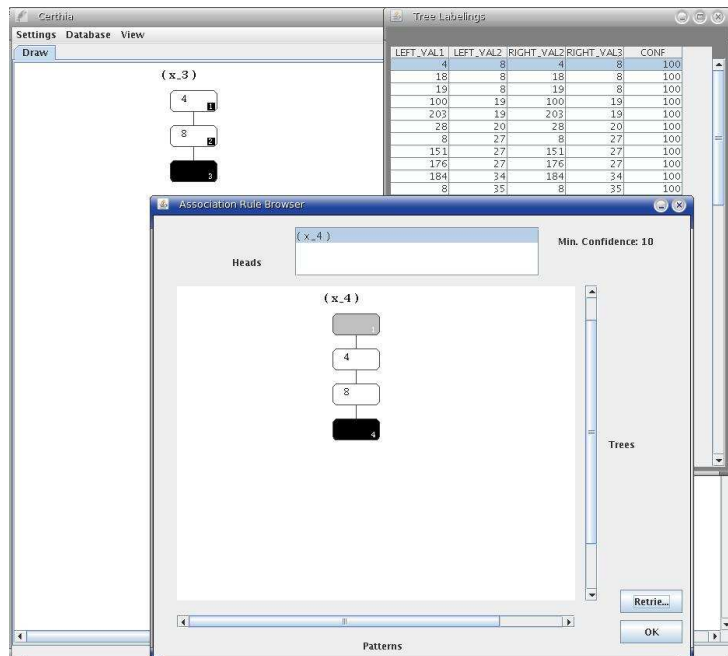


Figure 28: Screenshot of Certhia: the user asks for all association rules for this lhs.

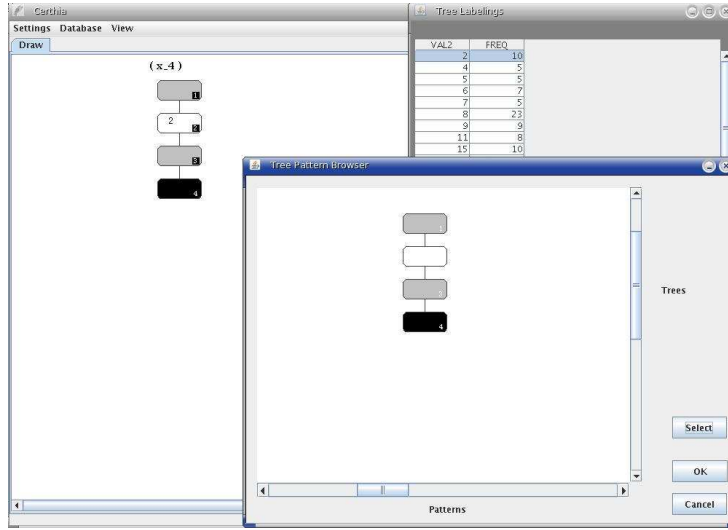


Figure 29: Screenshot of Certhia: the browser suggests frequent tree patterns.

a few hours, as long as after that the interactive exploration of the found results can happen very fast. And indeed we found that the actual generation of association rules is very fast. This is also shown in Section 7.2.

7 Experimental Results

In this section, we report on some experiments performed using our prototype implementation applied to both real-life and synthetic datasets to show that our approach is indeed workable.

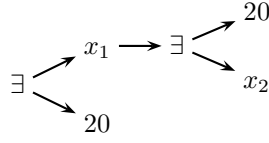
7.1 Real-life datasets

We have worked with a food web, a protein interactions graph, and a citation graph. For each dataset we built up a pattern database using the following parameters:

	#nodes	#edges	k	size
food web	154	370	25	6
proteins	2114	4480	10	5
citations	2500	350000	5	4

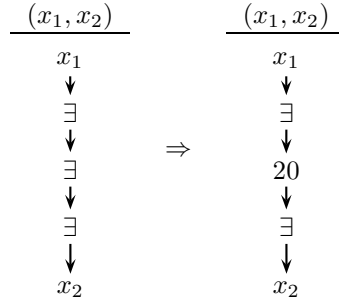
As we set rather generous limits on the maximum size of trees, or on the minimum frequency threshold, each run took several hours.

The **food web** [34] comprises 154 species that are all directly or indirectly dependent on the Scotch Broom (a kind of shrub). One of the patterns that was mined with frequency 176 is the following:



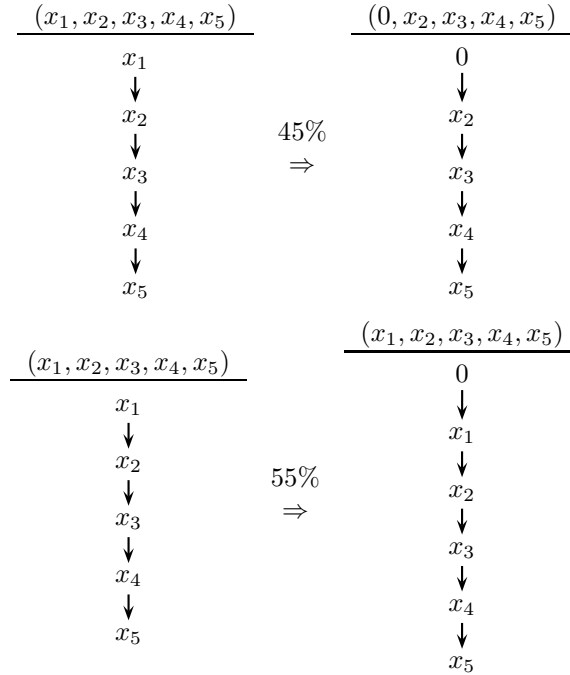
This is really a rather arbitrary example, just to give an idea of the kind of complex patterns that can be mined. Note also that, thanks to the constant 20 appearing twice, this is really a non-tree shaped pattern: we could equally well draw both arrows to a single node labeled 20.

While we were thus browsing through the results, we quickly noticed that the constant 20 actually occurs quite predominantly, in many different frequent patterns. This constant denotes the species *Orthotylus adenocarp*i, an omnivorous plant bug. To confirm our hypothesis that this species plays a central role in the food web, we asked for all association rules with the following left-hand side:



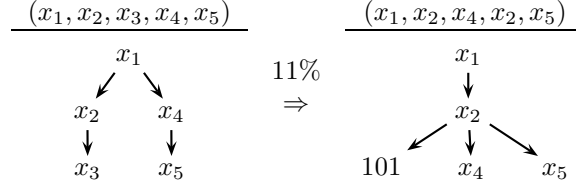
Indeed, the rule shown above turned up with 89% confidence! For 89% of all pairs of species that are linked by a path of length four, *Orthotylus adenocarp*i is involved in between.

Two other rules we discovered are:

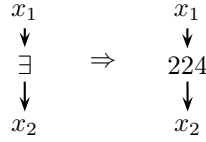


Since $45\% + 55\% = 100\%$, these rules together say that each path of length 5 either starts in 0, or one beneath 0. This tells us that the depth of the food web equals 6. Constant 0 turns out to denote the Scotch Broom itself, which is the root of the food web.

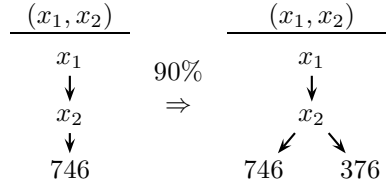
Another rule we mined, just to give a rather arbitrary example of the kind of rules we find with our algorithm, is the following:



The **protein interaction graph** [25] comprises molecular interactions (symmetric) among 1870 proteins occurring in the yeast *Saccharomyces cerevisiae*. In such interaction networks, typically a small number of highly connected nodes occur. Indeed, we discovered the following association rule with 10% confidence, indicating that protein #224 is highly connected:

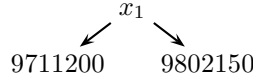


We also found the following rule:

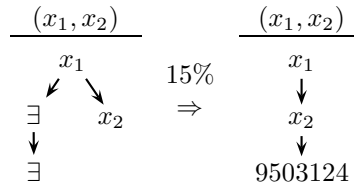


This rule expresses that almost all interactions that link to protein 746 also link to protein 376, which unveils a close relationship between these two proteins.

The **citation graph** comes from the KDD cup 2003, and contains around 2500 papers about high-energy physics taken from arXiv.org, with around 350 000 cross-references. One of the discovered patterns is the following, with frequency 1655, showing two papers that are frequently cited together (by 6% of all papers).



One of the discovered rules is the following:



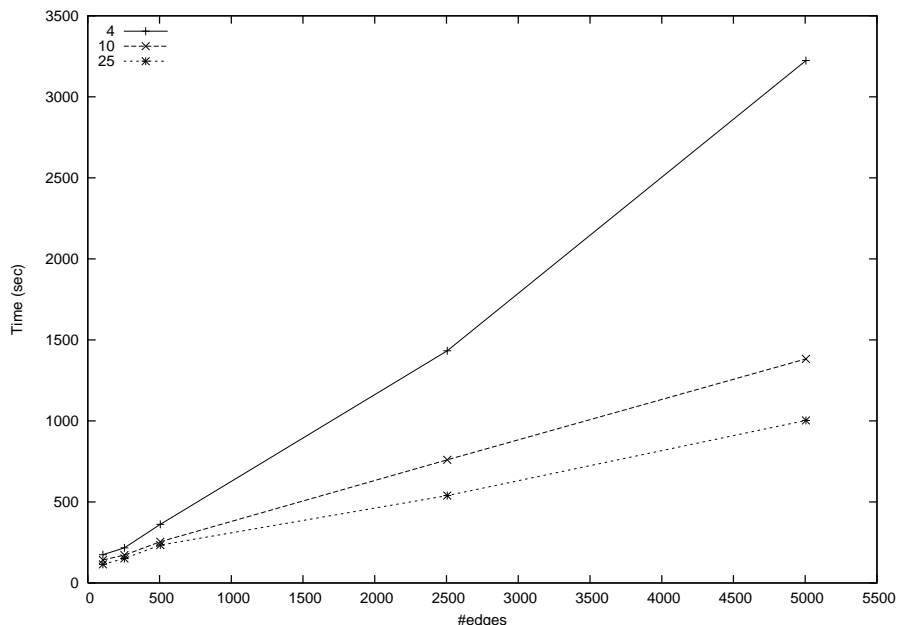


Figure 30: Performance on Web graphs.

This rule shows that paper 9503124 is an important paper. In 15% of all “non-trivial” citations (meaning that the citing paper cites at least one paper that also cites a paper), the cited paper cites 9503124.

7.2 Performance

While our prototype implementations are not tuned for performance, we still conducted some preliminary performance measurements, with encouraging results. The experiments were performed on a Pentium IV (2.8GHz) architecture with 1GB of internal memory, running under Linux 2.6. The program was written in C++ with embedded SQL, with DB2 UDB v8.2 as the relational database system.

We have used two types of synthetic datasets.

Random Web graphs Naturally occurring graphs (as found in biology, sociology, or the WWW) have a number of typical characteristics, such as sparseness and a skewed degree distribution [35]. Various random graph models have been proposed in this respect, of which we have used the “copy model” for Web graphs [27, 6]. We use degree 5 and probability $\alpha = 10\%$ to link to a random node (thus 90% to copy a link).

On these graphs, we have measured the total running time of the tree query mining algorithm as a function of the size (number of edges) of the graph, where we mine up to tree size 5, with varying minimum frequency thresholds of 4, 10, and 25. The results, depicted in Figure 30, show that the performance of these runs is quite adequate.

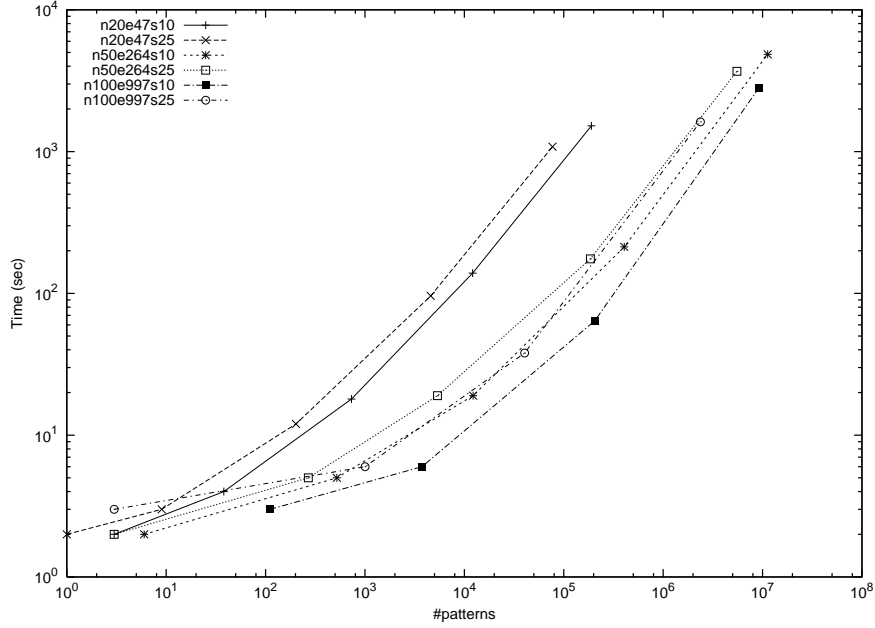


Figure 31: Performance in terms of number of discovered patterns.

Uniform random graphs We have also experimented with the well-known Erdős-Rényi random graphs, where one specifies a number n of nodes and gives each of the possible n^2 edges a uniform probability (we used 10%) of actually belonging to the graph. In contrast to random Web graphs, these graphs are quite dense and uniform, and they serve well as a worst-case scenario to measure the performance of the tree query mining algorithm as a function of the number of discovered patterns, which will be huge.

We have run on graphs with 47, 264, and 997 edges, with minimum frequency thresholds of 10 and 25. The results, depicted in Figure 31, show, first, that huge numbers of patterns are mined within a reasonable time, and second, that the overhead per discovered pattern is constant (all six lines have the same slope).

On these uniform random graphs we also conducted some experiments to check the performance of the association rule mining algorithm. We found the actual generation of association rules (i.e., loops 3 and 4, assuming that a pattern database is already build up) to be very fast. For instance, Figure 32 shows the performance of generating association rules for two different (absolute) values of minconf, against a frequency table database built up for a random graph with 33 nodes and 113 edges, an absolute minsup of 25, and all trees up to size 7. We see that associations are generated with constant overhead, i.e., in linear-output time. The coefficient is larger for the larger minconf, because in this experiment we have counted instantiated rhs's, and per rhs query less instantiations satisfy the confidence threshold for larger such thresholds. Had we simply counted rhs's regardless of the number of confident instantiations, the two lines would have had the same slope.

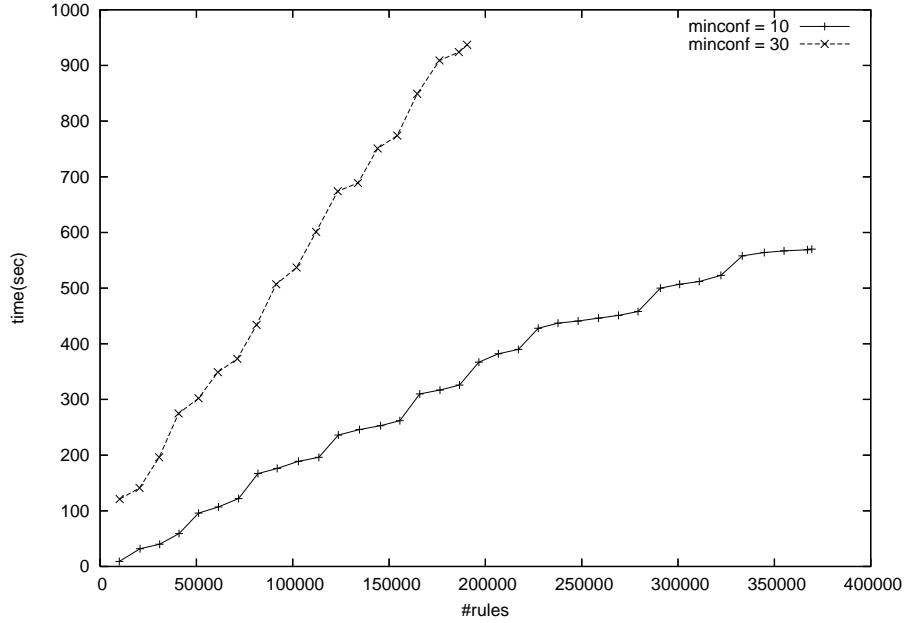


Figure 32: Performance in terms of number of discovered rules.

Performance issues One major performance issue that we have not addressed in the present study is that some of the SQL queries that are performed due to pattern generation take a very long time (in order of hours) to answer by the database system. This happens in those cases where the data graph is large (5000 edges or more) with many cycles, and the candidate patterns are large (6 nodes or more). Certainly, some SQL queries can be hand-optimized (or replaced by a combination of simpler queries), to alleviate these performance problems, but we leave this issue to future research.

Acknowledgment

We thank Bart Goethals for his contributions in the initial conception of the ideas presented in this work. We also thank Jan Hidders and Dries Van Dyck for their help with our results on graph isomorphism.

References

- [1] *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, volume 27:2 of *SIGMOD Record*. ACM Press, 1998.
- [2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- [3] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. Fast discovery of association rules. In Fayyad et al. [14], pages 307–328.

- [4] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [5] A. Amir, R. Feldman, and R. Kashi. A new and versatile method for association generation. *Information Systems*, 2:333–347, 1997.
- [6] K. Bharat, B.-W. Chang, M. Henzinger, and M. Ruhl. Who links to whom: Mining linkage between Web sites. In N. Cercone, T.Y. Lin, and X. Wu, editors, *Proceedings of the 2001 IEEE International Conference on Data Mining (ICDM 2001)*, pages 51–58. IEEE Computer Society Press, 2001.
- [7] S. Brin, R. Motwani, J.D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, volume 26:2 of *SIGMOD Record*, pages 255–264. ACM Press, 1997.
- [8] S. Chakravarthy, R. Beera, and R. Balachandran. DB-Subdue: Database approach to graph mining. In H. Dai, R. Srikant, and C. Zhang, editors, *Advances in Knowledge Discovery and Data Mining, Proceedings 8th PAKDD Conference*, volume 3056 of *Lecture Notes in Computer Science*, pages 341–350. Springer, 2004.
- [9] A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings 9th ACM Symposium on the Theory of Computing*, pages 77–90. ACM Press, 1977.
- [10] Yun Chi, Yirong Yang, and Richard R. Muntz. Canonical forms for labelled trees and their applications in frequent subtree mining. *Knowl. Inf. Syst.*, 8(2):203–234, 2005.
- [11] D.J. Cook and L.B. Holder. Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research*, 1:231–255, 1994.
- [12] L. Dehaspe and H. Toivonen. Discovery of frequent Datalog patterns. *Data Mining and Knowledge Discovery*, 3(1):7–36, 1999.
- [13] Luc Dehaspe and Hannu Toivonen. Discovery of relational association rules. In Saso Dzeroski and Nada Lavrac, editors, *Relational Data Mining*, pages 189–212. Springer-Verlag, 2001.
- [14] U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. MIT Press, 1996.
- [15] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *Proceedings of International Workshop on Graph-based Representation in Pattern Recognition, Ischia, Italy*, 2001.
- [16] S. Ghazizadeh and S. Chawathe. SEuS: Structure extraction using summaries. In S. Lange, K. Satoh, and C.H. Smith, editors, *Discovery Science*, volume 2534 of *Lecture Notes in Computer Science*, pages 71–85. Springer, 2002.

- [17] B. Goethals, E. Hoekx, and J. Van den Bussche. Mining tree queries in a graph. In Robert L. Grossman, Roberto Bayardo, Kristin Bennett, and Jaideep Vaidya, editors, *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.*, pages 61–69. ACM, 2005.
- [18] B. Goethals and J. Van den Bussche. Relational association rules: getting warmer. In D. Hand, R. Bolton, and N. Adams, editors, *Proceedings of the ESF Exploratory Workshop on Pattern Detection and Discovery in Data Mining*, volume 2447 of *LNCIS*, pages 125–139. Springer-Verlag, 2002.
- [19] Ehud Gudes, Solomon Eyal Shimony, and Natalia Vanetik. Discovering frequent graph patterns using disjoint paths. *IEEE Transactions on Knowledge and Data Engineering.*, 18(11):1441–1456, 2006.
- [20] E. Hoekx and J. Van den Bussche. Mining for tree-query associations in a graph. In *Proceedings of the 6th IEEE International Conference on Data Mining (ICDM 2006)*, pages 254–264. IEEE Computer Society, 2006.
- [21] Tamás Horváth, Jan Ramon, and Stefan Wrobel. Frequent subgraph mining in outerplanar graphs. In Tina Eliassi-Rad, Lyle H. Ungar, Mark Craven, and Dimitrios Gunopulos, editors, *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 197–206. ACM, 2006.
- [22] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM 2003)*, pages 549–552. IEEE Computer Society Press, 2003.
- [23] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. A general framework for mining frequent subgraphs from labeled graphs. *Fundamenta Informaticae*, 66(1-2):53–82, 2005.
- [24] G. Jeh and J. Widom. Mining the space of graph properties. In W. Kim, R. Kohavi, J. Gehrke, and W. DuMouchel, editors, *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 187–196. ACM Press, 2004.
- [25] H. Jeong, S.P. Mason, et al. Lethality and centrality in protein networks. *Nature*, 411(3 May 2001).
- [26] R.J. Bayardo Jr. Efficiently mining long patterns from databases. In *SIGMOD Conference* [1], pages 85–93.
- [27] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal. Random graph models for the web graph. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 57–65. IEEE Computer Society, 2000.
- [28] Michihiro Kuramochi and George Karypis. An efficient algorithm for discovering frequent subgraphs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1038–1051, 2004.

- [29] Michihiro Kuramochi and George Karypis. Finding frequent patterns in a large sparse graph^{*}. *Data Mining and Knowledge Discovery*, 11(3):243–271, 2005.
- [30] G. Li and F. Ruskey. The advantages of forward thinking in generating rooted and free trees. In *Proceedings 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 939–940, 1999.
- [31] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1(3):241–258, 1997.
- [32] B. D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.
- [33] Brendan D. McKay. nauty user’s guide, version 2.2. <http://cs.anu.edu.au/bdm/nauty/nug.pdf>.
- [34] J. Memmott, N.D. Martinez, and J.E. Cohen. Predators, parasites and pathogens: species richness, trophic generality, and body sizes in a natural food web. *Journal of Animal Ecology*, 69:1–15, 2000.
- [35] M. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.
- [36] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. *Data Mining and Knowledge Discovery*, 4(2–3):89–125, 2000.
- [37] H.I. Scions. Placing trees in lexicographic order. In D. Michie, editor, *Machine Intelligence 3*, pages 43–62. Edinburgh University Press, 1968.
- [38] W.-M. Shen, K. Ong, B.G. Mitbender, and C. Zaniolo. Metaqueries for data mining. In Fayyad et al. [14], pages 375–398.
- [39] S. Tsur, J.D. Ullman, et al. Query flocks: A generalization of association-rule mining. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data* [1], pages 1–12.
- [40] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume II. Computer Science Press, 1989.
- [41] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002)*, pages 721–724. IEEE Computer Society Press, 2002.
- [42] Mohammed J. Zaki. Efficiently mining frequent trees in a forest: Algorithms and applications. *IEEE Trans. Knowl. Data Eng.*, 17(8):1021–1035, 2005.

Appendix

Notation	Interpretation
U	set of data constants
T	ordered rooted tree
G	data graph
P	parameterized tree pattern
Π	set of existential nodes
Δ	set of distinguished nodes
Σ	set of parameters
\exists	existential node
σ	parameter
x	distinguished node
α	parameter assignment
$P^\alpha, (P, \alpha)$	instantiated tree pattern
$P^\alpha(G)$	$\{\mu _\Delta : \mu \text{ is a matching of } P^\alpha \text{ in } G\}$
minsup	the frequency threshold
$Q = (H, P)$	parameterized tree query with H the head and P the body
$Q^\alpha, (Q, \alpha)$	instantiated tree query
$Q^\alpha(G)$	answer set of the instantiated tree query Q^α in G
ρ	parameter correspondence
$Q_2 \subseteq_\rho Q_1$	Q_2 is ρ -contained in Q_1
$\text{freeze}_\beta(P)$	the freezing of a tree pattern P
pAR	parameterized association rule
iAR	instantiated association rule
$Q_1 \Rightarrow_\rho Q_2$	pAR from Q_1 to Q_2
$(Q_1 \Rightarrow_\rho Q_2, \alpha)$	iAR from Q_1 to Q_2
minconf	the confidence threshold
$\text{Freq}(P^\alpha)$	the frequency of P^α in G
(Π, Σ)	a parameterized tree pattern P based on a fixed tree T
(Π, Σ, α)	an instantiated tree pattern P based on a fixed tree T
$\text{CanTab}_{\Pi, \Sigma}$	$\{\alpha \mid P^\alpha \text{ is a candidate instantiated tree pattern}\}$
$\text{FreqTab}_{\Pi, \Sigma}$	$\{\alpha \mid P^\alpha \text{ is a frequent instantiated tree pattern}\}$
δ	answer set correspondence
$P_1 \equiv_\rho^\delta P_2$	P_1 is (δ, ρ) -equivalent with P_2
$P_2^{\alpha_2}(G) \circ \delta$	$\{f \circ \delta : f \in P_2^{\alpha_2}(G)\}$
$P_1 \cong P_2$	P_1 and P_2 are isomorphic